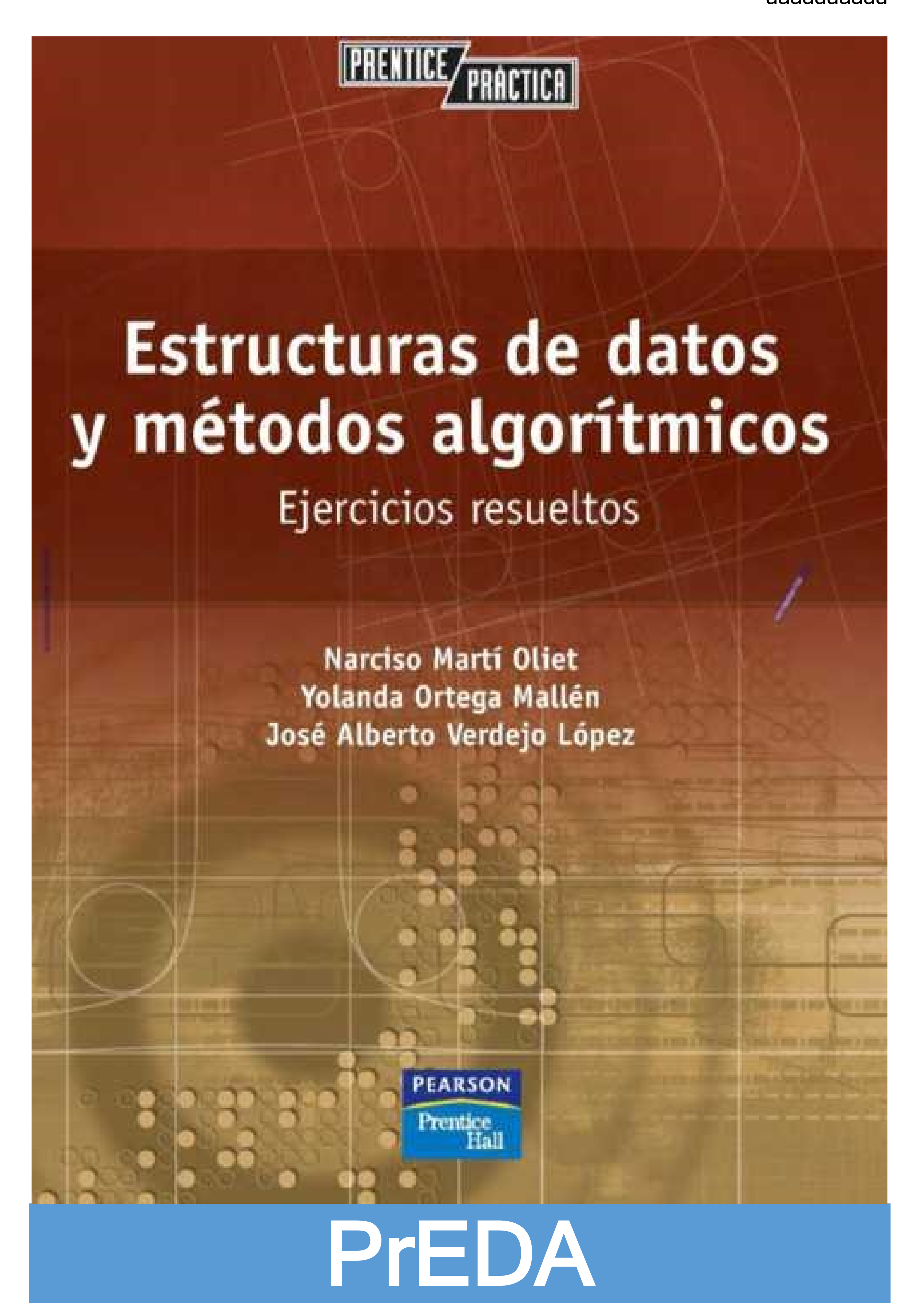
aaaaaaaaaa



PrEDA

PRÓLOGO

Este libro intenta llenar un hueco en el panorama editorial dedicado a libros académicos en el campo de la Informática. Si bien existen varios libros “de teoría" dedicados a algoritmos o estructuras de datos (muchos de ellos traducciones de originales en inglés), la cantidad de libros “de problemas" sobre estos temas es muy escasa, donde nos referimos a libros de *problemas resueltos.* En nuestra experiencia como profesores de asignaturas en las que se imparten estas materias, los alumnos demandan este material académico como complemento necesario a los otros libros que los profesores les aconsejamos sobre el tema, así como a los propios apuntes de clase. Una situación semejante se les plantea a profesores de asignaturas equivalentes con los que hemos hablado de este asunto. A lo largo de los años de enseñanza de estas materias hemos ido acumulando una serie de soluciones detalladas a varios ejercicios que en cierto momento pensamos era suficiente para satisfacer esa demanda por parte de los alumnos de estas asignaturas. Y de ahí surgió el planteamiento de este libro como colección de ejercicios resueltos sobre estructuras de datos y métodos algorítmicos.

El libro se divide en dos partes: la primera se dedica a las estructuras de datos y la segunda a los métodos algorítmicos, si bien es imposible separar por completo ambos, y esto se nota en algunos de los ejercicios que se incluyen. Cada parte se divide en una serie de capítulos dedicados a los temas típicos de cada materia Específicamente, la parte de estructuras de datos empieza con sendos capítulos genéricos sobre especificación algebraica de tipos abstractos de datos y su implementación, para continuar con capítulos dedicados a las estructuras de datos más habituales: pilas, colas, listas, árboles binarios y generales, árboles de búsqueda y tablas, colas con prioridad y montículos, y gratos. Esta parte

termina con un capítulo de aplicaciones donde se usan las estructuras anteriores de diferentes formas. Por otro lado, la parte de algoritmos se organiza alrededor de la clasificación habitual de métodos algorítmicos: divide y vencerás, método voraz, programación dinámica, vuelta atrás y ramificación y poda.

Cada capítulo empieza con una breve introducción al tema correspondiente, bien estructura de datos o bien método algorítmico. Sin embargo, tal introducción solamente intenta fijar los conceptos y notaciones que se utilizan en el desarrollo posterior de las soluciones de los ejercicios y no supone un tratamiento detallado del tema, para el cual remitimos a algunos de los libros de teoría de los cuales hay varios a disposición del alumno o profesor, incluyendo los que mencionamos en la bibliografía. A continuación aparece un listado de los enunciados de los ejercicios del tema, cada uno seguido de la solución correspondiente. Aunque la intención del libro es proporcionar soluciones detalladas a todos los ejercicios propuestos, animamos al estudiante y al lector en general a que. antes de mirar la solución, dedique cierto tiempo a comprender lo que se pregunta en el ejercicio y a intentar resolverlo por sí mismo para después comparar su solución con la propuesta. En general, la solución que se ofrece no es la única posible en absoluto y de hecho en algunos ejercicios presentamos más de una solución cuando consideramos que la variación es suficientemente interesante.

Muchos de los ejercicios que se plantean en el libro aparecen propuestos en otros libros de texto (Brassard y Bratley [BB90. BB97], Franch [Fra94], Horowitz y Sahni [HS94. HSR98], Manber (Man89], Parberry |Par95], Peña |Peñ98]). Algunos de estos problemas son tan famosos que aparecen repetidos en la mayoría de estos textos (por ejemplo, el problema de la mochila, el problema del viajante, etc.). En otros casos, hemos escogido enunciados de nuestros propios listados de ejercicios y de exámenes para las asignaturas que hemos impartido. También hemos

tomado prestados otros enunciados que han llegado a nuestras manos, en muchos casos de profesores compañeros a quienes estamos muy agradecidos por su amabilidad en compartirlos (María Inés Fernández, David de Frutos. Antonio Gavilanes, Luis Llana. Chus Martín. Ricardo Peña, Mario Rodríguez). En todos los casos consideramos que los enunciados de los ejercicios pertenecen de hecho al dominio público y no pretendemos tener ningún derecho exclusivo sobre ellos. Nuestra aportación original consiste en la redacción detallada de las soluciones a tales ejercicios, incluyendo los programas (en un lenguaje abstracto pero suficientemente cercano a lenguajes de programación imperativos) y no solamente las ideas o esbozos de muy alto nivel.

A la hora de escribir las especificaciones y los programas se nos planteó el eterno dilema de qué lenguaje elegir para ello. Nuestra decisión como profesores, y que hemos seguido también en este libro, es utilizar un lenguaje abstracto para ambos aspectos, de forma que se evitan algunos detalles demasiado concretos que suponen la utilización de un lenguaje particular y se puede poner el énfasis de las soluciones en un nivel de abstracción superior. Creemos que esta decisión tiene dos ventajas adicionales: por una parte, hace que el libro sea fácilmente adaptable para asignaturas en las que se utiliza un lenguaje de programación concreto: por otra, para pasar de una solución en el lenguaje abstracto a un programa en un lenguaje específico se requiere todavía cierto trabajo que se deja en manos del estudiante, de manera que no se le da todo hecho. Este paso se puede dar. por ejemplo, en las asignaturas de laboratorio que normalmente complementan las asignaturas teóricas hacia las que se enfoca principalmente este texto.

AGRADECIMIENTOS

Muchos han sido los compañeros que a lo largo de estos años de trabajo como profesores en la Universidad Complutense nos han prestado apoyo y ayuda en la preparación de las clases en general y la resolución de ejercicios en particular. Pero

desde estas líneas queremos agradecer con especial cariño a David de Frutos todo su esfuerzo y dedicación hacia nosotros en tan vanadas facetas, desde asesor científico hasta corrector de estilo, pasando por compañero y amigo. Sus críticas a este trabajo han sido numerosas, pero siempre constructivas, y el lector avezado sabrá apreciar el buen aprovechamiento que de ellas hemos hecho.

Queremos agradecer también a María Inés Fernández. Ricardo Peña y Mario Rodríguez sus valiosos consejos y la experiencia docente que han puesto en todo momento a nuestra disposición.

La versión actual de este libro se ha beneficiado enormemente de los "aparentemente inofensivos" comentarios y detalladas correcciones a múltiples versiones previas de David de Frutos. Miguel Palomino y Clara Segura. También hemos recibido acertados e interesantes comentarios de Francisco Duran. Luis Llana. Ricardo Peña e Isabel Pita. A todos ellos queremos darles las gracias por su contribución y esfuerzo.

Nuestro agradecimiento se hace extensivo asimismo a todos los alumnos que a lo largo de los últimos diez años han soportado nuestras clases, nos han obligado a esforzarnos en nuestras explicaciones y varias veces han sugerido mejoras.

David Fayerman y Miguel Martín-Romo se merecen nuestras gracias por su acogida de nuestra propuesta para publicar este libro en la colección *Prentice Practica* y por la libertad que nos han concedido para la realización del proyecto.

Y llegamos al turno de agradecimientos familiares. Puede que suene a tópico aquello de *debemos agradecer a nuestras familias que hayan sufrido nuestro estrés y nuestra mermada dedicación a ellos en este último año,* pero es rigurosamente cierto. Padres, cónyuges, hermanos, hijos, cuñados, amigos, etc., todos saben de la existencia del "lamoso libro" y han anhelado su pronta terminación. No porque

tuvieran interés alguno en leerlo, sino con la esperanza de no seguir viéndonos encadenados a nuestros portátiles. Lo que no imaginan ellos es que ahora tenemos un montón de trabajo atrasado pendiente.

Para terminar, agradecemos a la madre de Alberto sus exquisitas empanadillas dulces que en tantas ocasiones nos han dado energía suficiente para aguantar sesiones maratonianas de discusión sobre el libro. Y a Hernán Cortés por haber traído el chocolate a Europa, porque en realidad nunca nos reunimos para escribir un libro, sino para realizar sesiones de degustación de chocolate.

Los autores, en Madrid, julio 2003.

PARTE 1: ESTRUCTURAS DE DATOS

Capítulo 1

1. ESPECIFICACIÓN ALGEBRAICA DE TIPOS ABSTRACTOS DE DATOS

1.1. INTRO

Con ocasión del último cambio de milenio en el calendario occidental tuvo lugar un fenómeno generalizado de "histeria informática" conocido como el *problema del año 2000.* debido a la necesidad de adaptar equipos y programas informáticos para poder trabajar con fechas en las que el año se representa con los 4 dígitos en vez de solamente con los dos últimos. Por supuesto, como dijo el profesor Donald E. Knuth, el problema se agudizará cuando se acerque el año 10000. pero en ambos casos, y simplificando muchas cosas, podemos decir que. desde el punto de vista de la programación, se trata de un problema de falta de *abstracción,* al no diferenciar entre la representación y el uso de un tipo de datos (en este caso,

fechas). Cualquier cambio en la representación de los datos, motivado por la limitación de esta, nos obliga a buscar todos los lugares de los programas donde aparecen esos datos, para allí modificar la misma.

1.1.1 Tipos abstractos de datos

La idea fundamental de los ***tipos abstractos de datos*** (TADs) es separar de forma estricta la representación del uso de los datos de un tipo. Para empezar hemos de tener en cuenta que un tipo no solamente consta de un conjunto de valores, sino también de un conjunto de operaciones para la creación, modificación y manipulación de dichos valores, a través de las cuales se realizará el uso de los datos.

Por una parte, tenemos la representación del tipo de datos en términos de tipos básicos o de otros tipos ya conocidos, así como la implementación sobre dicha representación de las operaciones asociadas al tipo de datos, que constituyen la *interfaz* de ese tipo. En los algoritmos que implementan esas operaciones se tiene pleno acceso a la representación concreta del tipo de datos. Así, en el ejemplo de las fechas se puede ver si el año se representa con 2 o con 4 dígitos, o de otra forma.

Por otra parte, la única forma de usar los datos del tipo abstracto es invocando adecuadamente las operaciones de la interfaz, de manera que nunca se accede

a su representación interna, de ahí la abstracción. Así podemos tener diferentes representaciones para un mismo tipo de datos, pero cuando se usa el tipo nunca se sabe cuál es la representación que se está utilizando. Cuando se cambia la representación, siempre y cuando la interfaz se mantenga, los programas que usan el tipo deben continuar funcionando exactamente de la misma forma que antes.

En particular, una vez se ha fijado la interfaz del tipo, la programación de la representación del tipo e implementación de las operaciones por un lado, y la programación de los algoritmos que usan el tipo por otro, pueden proceder de manera completamente independiente la una de la otra. Sin embargo, para que esto sea posible sin que los programadores de los segundos conozcan la representación concreta que manejan los programadores de las primeras, hace falta que intercambien no solamente la interfaz sino también una descripción apropiada del comportamiento de las operaciones que constituyen dicha interfaz.

Estas ideas resultarán sin duda familiares para aquellos lectores que ya conozcan la programación orientada a objetos, que históricamente constituye un desarrollo posterior y que ha incorporado el concepto de abstracción estudiado para los tipos de datos.

Entre muchas propuestas que existen

para *especificar* el comportamiento de las operaciones sobre un tipo de datos, vamos a considerar en este libro la conocida como *especificación algebraica* o *ecuacional.* que se basa en describir el comportamiento mediante ecuaciones, lo cual facilita el estilo habitual de razonamiento ecuacional, basado en sustituir iguales por iguales.

1.1.2. Especificación algebraica con constructoras

Un tipo abstracto de datos viene determinado por las operaciones asociadas, incluyendo constantes que se consideran como operaciones con cero argumentos.

Entre estas operaciones vamos a distinguir unas que denominamos *constructoras* (en algunos textos se las denomina *generadoras)* que sirven para construir o generar lodos los datos del tipo que se está especificando. Para un mismo tipo de datos, como veremos, puede haber distintas elecciones de constructoras. Según las constructoras elegidas, puede ser que haya una única forma de representar con esas constructoras cada dato, en cuyo caso decimos que las constructoras son *libres,* o que haya diferentes representaciones para un mismo dato. En este último caso, es necesario identificar las razones por las que hay varias formas de decir lo mismo, proporcionando una serie de *ecuaciones de equivalencia* entre

términos construidos.

Las restantes operaciones de la interfaz del tipo de datos se clasifican en *observadoras* y *modificadoras.* Las primeras son aquellas que devuelven información sobre los datos que se están especificando, y esta información pertenece a otros tipos de datos ya conocidos. Las segundas son operaciones cuyo resultado es un dato del mismo tipo que se especifica.

El comportamiento de las operaciones observadoras y modificadoras se especifica mediante *ecuaciones,* que pueden ser condicionales. La *metodología de constructoras* consiste en que, una vez elegidas las constructoras para el tipo de datos, las ecuaciones definen el comportamiento de las operaciones observadoras y modificadoras en función de su efecto sobre los *términos construidos,* o sea. sobre los términos que se definen sobre variables tipadas apropiadamente y las operaciones constructoras. Entonces, para una operación observadora o modificadora op cuyos datos de entrada tienen tipos 7j *.* T_n*.* las ecuaciones que definen su comportamiento tienen la forma Op(Cj, C_2 C_n) = t_0 <= Z| = ri A ... A t_k = *ti*.

donde los términos *ci* son términos construidos de tipo *T,* para *i* entre 1 y *n.* mientras que z_0 y *tj.* z' (con *j* entre 1 y *k,* y *k* > 0) son términos que además de las constructoras pueden involucrar también

la propia operación op (dando lugar a definiciones de carácter recursivo) y otras operaciones modificadoras u observadoras.

Nótese que la condición de la ecuación anterior es una conjunción (que puede ser vacía) de ecuaciones. Dado un *predicado* P. es decir, una operación con perfil P : *T* 7? ... *T„ —> bool.* donde *bool* es el tipo usual de los *valores booleanos* (véase la Sección 1.1.4), abreviaremos una condición de la forma P(ri,..., t_n) = cierto como P(z j,..., t_n), y una condición de la forma P(ri, .. , t_n) = falso como ~'P(/j t_n), donde —» denota la negación sobre booleanos. Más cn general, una ecuación de la forma
t = cierto, donde *i* es un término de tipo *bool.* se abrevia como *t* en las condiciones.

Las operaciones que constituyen la interfaz de un tipo de datos, incluyendo las constructoras, pueden ser *parciales.* Esta situación se explicitará en la especificación señalando por un lado las operaciones parciales (mediante un subíndice *p* en el perfil de la operación), y escribiendo por otro lado *ecuaciones de error* que indican en qué situación la operación en cuestión no está definida. Sin embargo, no haremos un tratamiento explícito de errores, es decir, que vamos a suponer implícitamente que cualquier operación aplicada a un error devuelve un error, y no vamos a escribir ecuaciones que hagan explícita esta propagación de

errores.

1.1.3. Lenguaje abstracto de especificacion

Presentamos a continuación un esquema de especificación de tipos abstractos de datos para introducir la notación que vamos a seguir en este libro. Se trata de un *lenguaje abstracto de especificación* que no corresponde exactamente a ningún lenguaje concreto de especificación entre los muchos propuestos en la literatura sobre este lema, si bien guarda cierto parecido general con algunos de ellos.

especificación *NOMBRE-ESPEC*
usa *BOOLEANOS, OTRA-ESPEC. OTRA-MÁS*
tipos *nombre-tipo*
operaciones
c : —>*nombre-tipo*
f*nombre-tipo^...nombre-tipo,,* —> *nombre-tipo*
g*nombre-tipo}...nombre-tipo,,,* —*otro-tipo*
operaciones privadas
p*nombre-tipo^...nombre-tipo/* —> *bool*
h*nombre-tipo^* . . *nombre-tipo^* —> *otro-tipo*
variables
x, y : *nombre-tipo*
Z : otro-tipo
ecuaciones
/i = [2
zi = t'2<=C}=c\

fespecificación

En primer lugar, tenemos la *signatura* o *interfaz* del tipo de datos, que consta de

- el nombre de la especificación.
- una posible importación de otras especificaciones de tipos dando sus nombres.
- la declaración del nombre *nombre-tipo* del tipo de datos que se define, que en casi todos los ejemplos es solamente uno pero en general pueden ser varios, y
- la declaración de las operaciones asociadas, cada una de ellas declarada con su perfil, es decir, los tipos de los datos de entrada y el tipo del resultado.

En el esquema anterior, c es una constante pues no tiene argumentos, f es una operación total y g es una operación parcial como indica el subíndice *p* junto a la flecha del perfil. Las dos primeras podrían ser constructoras o modificadoras, mientras que la tercera es una observadora pues su resultado tiene un tipo diferente al especificado.

Después de las operaciones de la interfaz aparece la declaración de *operaciones privadas* o auxiliares que son útiles en la especificación pero no se pueden usar fuera de la misma. En el esquema ejemplo tenemos un predicado p (pues el tipo de su resultado es *bool)* declarado como operación privada, junto a otra operación privada h.

Cuando sea conveniente o habitual, utilizaremos notación infija para las operaciones: en tal caso usaremos el signo de subrayado para indicar las posiciones de los argumentos en la declaración de la operación, como por ejemplo _ + _ para una operación binaria de suma.

Tras la signatura viene la declaración de variables, cada una con su tipo correspondiente, que se usarán en los términos para formar ecuaciones; y finalmente, la lista de ecuaciones, posiblemente condicionales. En el esquema ejemplo tenemos una ecuación sin condición y otra condicional.

Recordemos que las ecuaciones se pueden clasificar, tal como hemos explicado anteriormente, en:

- ecuaciones de equivalencia entre términos construidos, cuando las constructoras no son libres,
- ecuaciones de error, en caso de operaciones parciales, y
- ecuaciones que definen el comportamiento de las operaciones modificadoras y observadoras en función de las constructoras.

Debido a su sencillez y a ser bien conocido tanto en el contexto de la lógica como en el de los lenguajes de programación, consideraremos como primer ejemplo de especificación algebraica el tipo de datos de los *valores booleanos.*

especificación *BOOLEANOS*
tipos *bool*
operaciones
cierto :—> *bool*{constructora}
falso :—> *bool*{constructora)
: bool—> bool
_ A _ :*bool bool* —> *bool*
_ v _*bool bool* —> *bool*

Tenemos constantes cierto y falso, la operación linaria de negación y operaciones binarias de conjunción y disyunción.

Puesto que el tipo del resultado de todas las operaciones es *bool.* que es el tipo que se especifica, en este ejemplo no hay operaciones observadoras.

Entre las operaciones dadas en la

interfaz. tenemos que elegir un conjunto de constructoras l.as dos constantes obviamente representan de forma única los dos valores del tipo de datos que se define, por lo que constituyen un conjunto de constructoras apropiado, tal como se ha señalado en la signatura anterior. Al tratarse de dos constantes, el conjunto de términos que definen está constituido por esas dos mismas constantes, y como cada valor está representado exactamente por una de ellas, *no* hacen falta ecuaciones de equivalencia entre los términos construidos, es decir, las constructoras son *libres.*

Con esta elección de constructoras (que no es la única posible, como veremos en el Ejercicio 1.1). las otras tres operaciones son modificadoras y debemos definir su comportamiento en términos de las constructoras. Para ello consideramos, en primer lugar, la operación de conjunción, que es binaria, con dos argumentos de tipo *bool.* Como hay dos constructoras del tipo *bool.* en principio para cada uno de los argumentos tenemos dos posibilidades, con lo cual sería posible considerar cuatro ecuaciones de la forma siguiente:

cierto A cierto = cierto
cierto A falso = falso
falso A cierto = falso
falso A falso = falso

que representan la tabla de verdad de la conjunción. Sin embargo, no es

necesario distinguir tantos casos (cuando el número de constructoras se incrementa, este detalle adquiere más importancia a la hora de facilitar la lectura y comprensión de la especificación), pues basta con tener información sobre un argumento para saber el resultado, utilizando una variable *b* de tipo *bool* para representar de forma genérica el otro argumento:

cierto A *b* = *b*

falso A *b* = falso

En este ejemplo da igual el argumento sobre el cual empiezan a distinguirse los casos, pues el comportamiento de la conjunción es simétrico con respecto a los dos argumentos, es decir, se trata de una operación binaria conmutativa. Sin embargo, aunque esta propiedad sea cierta, *no* vamos a poner en la especificación una ecuación de conmutatividad de la forma

b /\ b' = b' f\b

que no sigue la metodología de constructoras, pues no define el comportamiento de la operación que se está especificando sobre valores del tipo de datos (dados como términos construidos). Razonando con las dos ecuaciones anteriores, se puede ver que la ecuación de conmutatividad es de hecho consecuencia de esas dos ecuaciones y es, en ese sentido, innecesaria en esta clase de especificaciones.

Las ecuaciones que especifican el comportamiento de la disyunción son parecidas a las de la conjunción y las de la negación son inmediatas, de forma que el resto de la especificación tiene la siguiente forma, donde las ecuaciones aparecen tras la declaración de las variables que se usan.

variables

b : bool **ecuaciones**

-■cierto = falso

—falso = cierto

cierto A *b = b*

falso A *b* = falso
cierto V *b* = cierto
falso v *b* = *b*

fespecificación

En esta especificación no hay ninguna ecuación condicional. Otra característica habitual de las especificaciones ecuacionales. que tampoco se aprecia en este ejemplo debido a su sencillez, es la definición recursiva de las operaciones, como veremos en la mayoría de los ejercicios que siguen.

Muchas de las especificaciones de tipos de datos habituales que iremos viendo en los ejercicios y los capítulos siguientes, como son pilas, colas, listas, conjuntos, etc . son *genéricas* o *paramétricas* con respecto a los datos que contienen, es decir, la especificación de la construcción genérica de datos que se define es completamente independiente de los elementos concretos que constituyen esos datos. En tales casos, la especificación está parametrizada con respecto a un *parámetro* que indica los requisitos que deben cumplir los datos concretos para que tenga sentido aplicarles la construcción paramétrica.

Aunque la estructura de un parámetro es similar a la de una especificación, y dentro de esta se usa como una especificación importada, el parámetro debe entenderse como un argumento formal de la especificación que se va a instanciar adecuadamente cuando se use el tipo de datos genérico. Por ejemplo,

una vez definidos los conjuntos en general, después podemos usar los conjuntos de booleanos. los conjuntos de naturales, los conjuntos de conjuntos de naturales, etc.

El primer ejemplo de parámetro que consideramos solamente exige la existencia de un tipo.

parámetro *ELEM* **tipos** *elemento*
fparámetro

Aunque esto puede ser suficiente en algunas situaciones, en muchos casos interesa que el tipo tenga una operación de *igualdad* que permita comparar elementos entre sí; por ejemplo, si queremos poder decidir si dos conjuntos son iguales, debemos poder comparar los elementos que los forman para ver si

son iguales o no. El siguiente parámetro exige un tipo junto con operaciones de igualdad y desigualdad. Las ecuaciones *no* definen esas operaciones, sino que dan requisitos que deben cumplir para que en efecto las consideremos como tales. Específicamente, la desigualdad debe ser la negación de la igualdad, y la operación de igualdad tiene que decir que dos elementos son iguales si y solo si lo son realmente.

parámetro *ELEM=* **usa** *BOOLEANOS*
tipos *elemento* **operaciones**
  _ == _ *: elemento elemento —> bool*
      : elemento elemento —> bool
variables
  x. y : *elemento* **ecuaciones**
  (x == y) = cierto <= x = y
  x = y <= (x == y) = cierto
  A- / y = ->(x == y)
fparámetro

Nótese que x = v es una *ecuación* que afirma que los elementos x e y de tipo *elemento* son iguales, mientras que x == y es el *predicado de igualdad* aplicado a los dos elementos x e y. que da lugar a un término de tipo *bool:* entonces (x =— y) = cierto es una ecuación que afirma que el resultado de esa aplicación del predicado de igualdad es igual a la constante booleana cierto. Así. las dos primeras ecuaciones condicionales en el parámetro *ELEM—* afirman conjuntamente la equivalencia
(x == y) = cierto <=> x = y
según la cual la información que devuelve

la operación de igualdad refleja fielmente la información sobre el tipo de datos.

Esta equivalencia puede dar la falsa impresión de que nunca vamos a necesitar la operación de igualdad porque siempre podemos utilizar las ecuaciones del lenguaje abstracto de especificación en su lugar Cuando la igualdad devuelve cierto, tal y como aparece en la equivalencia anterior, efectivamente es así. pero cuando la igualdad devuelve falso la situación cambia porque en el lenguaje de especificación que vamos a usar en general no existe ningún recurso que permita escribir la "negación" de una ecuación (en las condiciones solamente pueden aparecen conjunciones de ecuaciones). En cambio, en presencia de un predicado de igualdad, como x == *y es* una expresión de tipo *bool* en el cual disponemos de la negación booleana, -’(x == y) es otra expresión del mismo tipo que intuitivamente afirma que x e y son distintos, lo cual, usando la definición en el parámetro *ELEM=,* podemos escribir como x / y y va a ser de gran utilidad a la hora de distinguir casos según que dos elementos sean iguales o distintos. De nuevo insistimos en que x y es un término de tipo *bool* obtenido al aplicar el predicado de desigualdad, de la misma forma que x == y es un término booleano obtenido al aplicar la igualdad, y en particular A y no es la negación de una ecuación, aunque la notación pudiera dar esa impresión. En general, la

existencia de una operación de igualdad sobre un tipo nos va a permitir escribir cualquier expresión booleana basada en ese predicado.

Aunque los dos parámetros anteriores. *ELEM* y *ELEM=.* son los más habituales en los ejercicios que siguen a lo largo de la primera parte del libro, un parámetro puede definirse de la forma adecuada a la especificación que se desee y veremos algunos otros ejemplos como órdenes totales, valores modilicables etc.

El nombre de una especificación parametrizada es de la forma *ESPECfPARAMJ* que indica el nom bre del parámetro requerido. Cuando se instancíe con un tipo de datos concreto, se llamará de la form; *ESPEC[CONCRETO],* Entonces las apariciones de los tipos y operaciones formales del parámetro se sustituyen por los tipos y operaciones concretos apropiados; muchas veces sobrecargaremos la notación de las operaciones, de forma que se llamen igual las operaciones concretas que las formales, con lo cual la sustitución es inmediata. En el caso de los nombres de tipos, haremos explícita la concreción, de manera que un nombre genérico como *lista* que indica listas en general se convierte en *lista[bool]* para listas de booleanos, por ejemplo.

(a) Extender la especificación de los valores booleanos vista en la introducción de este capítulo con

operaciones de implicación, equivalencia y disyunción exclusiva.

(b) Rehacer la especificación de los booleanos dada en la introducción cambiando el conjunto de constructoras.

Solución:

Apartado (a)

Extendemos la especificación *BOOLEANOS* (nótese la declaración usa) con la interfaz de las tres operaciones binarias adicionales y las ecuaciones correspondientes, distinguiendo casos sobre constructoras en el primer argumento y utilizando la operación de negación para obtener el resultado, en un par de casos.

especificación *BOOLEANOS+* **usa** *BOOLEANOS*

operaciones

_ => _*bool bool* —> *bool*

_ = _ :*bool bool* —> *bool*

_ xor _ :*bool bool* -—> *bool*

variables

b : bool **ecuaciones**

cierto => *b* = *b*

falso => *b* = cierto

cierto = *b* = *b*

falso = *b* = *--b*

cierto xor *b* = -•*b* falso xor *b* = *b*

fespecificación

Apartado (b)

Una posibilidad es considerar como constructoras una de las constantes (por ejemplo, cierto) y la negación. Entonces

hay infinitos términos construidos de la forma . . -■cierto.
Sabemos que el upo de datos consta tan solo de dos valores, uno que denota certeza y otro falsedad; el primero se representa con la constante cierto, mientras que el segundo se puede representar mediante el término -■cierto, por ejemplo. ¿Qué significan entonces los restantes términos construidos ?
Intuitivamente, negar dos veces anula el efecto de la negación, es decir, -■-■cierto debe comportarse como cierto, -■-■-'Cierto como -■cierto, etc. Por tanto, en esta especificación las constructoras *no son libres* y es necesario introducir *ecuaciones de equivalencia* entre los términos construidos, para lo cual basta la ecuación *-'-'b = b.* que indica que la doble negación se comporta como la identidad. Como cada valor se puede representar de múltiples formas, hay que asegurarse que las ecuaciones que definen las restantes operaciones (todas ellas modificadoras) son independientes del representante elegido.

Volvemos a escribir la signatura completa de la especificación, haciendo notar que el único cambio es la elección de constructoras, y a continuación incluimos el axioma de equivalencia y la definición de la constante falso.

especificación *BOOLEANOS+2* **tipos** *bool*
operaciones
cierto :—> *bool*{ constructora)

falso :—> *bool*
: *bool*—> *bool* { constructora }
_ A _ *:bool bool* —> *bool*
_ v _ :*bool bool* —> *bool*
_ => _ *:bool bool*—> *bool*
_ = _ *:bool bool* —> *bool*
_ xor _ :*bool bool*—> *bool*

variables

b, c : bool **ecuaciones**

-•-•*b* = *b* (doble negación }

falso = -'Cierto

Para especificar la conjunción y la disyunción, distinguimos casos sobre constructoras en el primer argumento. Cuando el primer argumento es cierto, tenemos suficiente información para conocer el resultado directamente, pero cuando es una negación, hemos de distinguir casos en el segundo argumento. Cuando el segundo argumento es cierto, damos el resultado directamente, y cuando es otra negación, nos apoyamos en las conocidas *leyes de De Morgan* que afirman que la conjunción/disyunción de negaciones es equivalente a la negación de la disyunción/conjunción. con lo que se reduce la conjunción inicial a una disyunción más sencilla, *y* viceversa. De esta forma obtenemos una definición *mutuamente recursiva* de ambas operaciones. La recursión está bien definida porque el tamaño de los correspondientes argumentos decrece estrictamente; concretamente, en la tercera ecuación de la conjunción y de la

disyunción los argumentos en la izquierda son los términos -'/> y *-•c.* mientras que en la derecha se reducen a los subtérminos *b* y *c.*

cierto A *c* = *c*

-•b A cierto = *-•b*

-<*b* A ->c = ->(b v c)

cierto ve = cierto

->*b* v cierto = cierto

-'b v *->c* = *—•(b* A c)

Para las tres últimas operaciones hacemos la misma distinción de casos, dando lugar a definiciones recursivas, pero no mutuamente recursivas.

cierto => *c* = *c*

-•*b* => cierto = cierto

->b => ->c = *c* => *b*

cierto = *c* = *c*

—'b = cierto = *->b*

-•*b* = ->c = *b* = *c*

cierto xor *c* = -•*c*

—•*b* xor cierto = *b*

-•*b* xor —= *b* xor *c*

Tanto en este apartado como en el anterior, en vez de distinguir casos sobre constructoras, podemos también escribir ecuaciones que definen unas operaciones lógicas en función de otras, de acuerdo con conocidas equivalencias lógicas. Por ejemplo.

b => *c* = (—-Z?) v *c*

b = *c* = *(b* => c) A (c => *b)*

b xor *c* = -•*(b* = *c)*

fespecificación

Especificar los *números naturales* con las siguientes operaciones:

- cero y sucesor,
- suma.
- producto,
- diferencia de naturales (al restar a un número otro mayor el resultado que se obtiene es cero),
- potencia,
- relaciones de igualdad y desigualdad,
- relaciones de orden <, <, > y >,
- máximo y mínimo.
- cociente y resto de la división entera, y
- predicados para reconocer si un natural es par o impar.

Solución:

La elección de constructoras no admite muchas posibilidades, pues con las operaciones dadas en el enunciado la única forma de generar un número natural es aplicar sucesivas veces la operación sucesor suc (que

intuitivamente suma uno) a la constante cero. Más concretamente, el número *n* se representado forma unívoca mediante el término construido suc"(cero) donde la notación suc" indica que la operación suc se aplica *n* veces. En definitiva, tenemos una representación *uñaría* (es decir, en base l) de los números naturales, de forma que para obtener el número *n* empezamos con el cero y sumamos uno *n* veces. Como la representación de cada valor del tipo definido es única, *no* hay ecuaciones de equivalencia entre los términos construidos, es decir, las constructoras cero y suc son *libres.*

A continuación vemos la interfaz de todas las operaciones que se piden en el enunciado. Nótese que los predicados de orden y de paridad son observadoras, pues el tipo de su resultado es *bool* y todas las demás, excepto las dos constructoras, son modificadoras. Por otra parte, todas las operaciones son totales, excepto div y mod que son *parciales,* debido a que no se puede dividir por 0.

especificación *NATURALES*
usa *BOOLEANOS*
tipos *nat*
operaciones
cero : —> *nat* (constructora)
suc*:* *nat* —>*nat* {constructora |
_ + _ *nat nat* —> *nat*
_ * _ *: nat nat* —> *nat*
_ — _ *: nat nat* —> *nat*
exp*:* *nat nat* —> *nat*

_ == _ *: nat nat —> bool*
_ *-.nat nat —> bool*
_ < _ *:natnat —> bool*
_ < _ *:natnat —> bool*
_ > _ *:natnat —> bool*
_ > _ *:natnat —> bool*
máx *: nat nat —> nat*
mín*:nat nat —> nat*
_ div _ *: nat nat —> $_p$ nat*
_ mod _ :*nat nat —> $_p$ nat*
es-par?*: nat —> bool*
es-impar? : *nat —> bool*

variables

n,m : nat

Ahora debemos especificar el comportamiento de las operaciones observadoras y modificadoras en términos de las constructoras. Empezamos por la suma, en cuya definición podemos aplicar las mismas ideas que ya vimos en la Sección 1.1.4 para la definición de la conjunción sobre los booleanos. En principio, dado que tenemos dos constructoras del tipo *nat.* podemos considerar dos posibilidades para cada uno de los argumentos de la suma, dando lugar a cuatro ecuaciones con casos cero-cero, cero- sucesor, sucesor-cero y sucesor-sucesor. Sin embargo, la especificación se puede simplificar reduciendo el número de ecuaciones al tener en cuenta que basta información sobre la constructora en uno de los argumentos para saber cómo realizar la suma; por ejemplo, si un argumento es cero, el resultado coincide

con el otro argumento.

Como la suma es conmutativa, es equivalente realizar la distinción de casos en uno u otro argumento y aquí lo hacemos sobre el primero. Si es cero, el resultado coincide con el segundo argumento, como ya hemos comentado. Si el primer argumento es un sucesor tenemos una definición recursiva, haciendo uso de la igualdad matemática *(n* + 1) + *ni = (/; + ni)* + 1. La recursión está bien definida porque el tamaño del primer argumento de la suma decrece estrictamente en la llamada recursiva, pasando de ser suc(n) a ser *n.*

ecuaciones

cero + *ni — m*

suc(n) + *ni* = suc(n + *ni)*

Intuitivamente, teniendo en cuenta la notación unaria, la interpretación recursiva de las dos ecuaciones anteriores se basa en que sumar *n* a *ni* es lo mismo que sumar *n* veces uno a *ni.*

Como ya hemos comentado para la conjunción booleana. no consideramos una ecuación de conmu- tatividad para la suma *n + ni = ni + n,* porque por una parte es una consecuencia de las dos ecuaciones anteriores, y por otra parte no sigue la metodología de constructoras que nos interesa.

Las consideraciones que acabamos de hacer para la distinción de casos en la especificación de la suma se aplican también al producto de números naturales: bastan dos ecuaciones en vez

de cuatro y no importa distinguir casos en uno u otro argumento por la conmutatividad. Cuando el primer argumento es cero, el resultado es también cero. Cuando el primer argumento es un sucesor, se hace una llamada recursiva, utilizando la igualdad matemática $(n + 1) * ni = (n * ni) + ni$. Obsérvese que utilizamos la suma, definida anteriormente, en la definición del producto. La recursión está bien definida por la misma razón que antes.

cero * *ni* = cero

suc(n) * *ni* = (n * *ni*) + *ni*

Intuitivamente, la interpretación recursiva de estas dos ecuaciones se basa en que multiplicar *n* por *ni* es lo mismo que sumar *n veces* la cantidad *ni.*

Con la resta la situación es algo diferente, al no ser una operación conmutativa, por lo que la distinción de casos no es simétrica con respecto a los dos argumentos. En las ecuaciones que siguen distinguimos primero casos en el primer argumento y cuando este es un sucesor distinguimos casos en el segundo. En las dos primeras ecuaciones, el resultado es inmediato, mientras que en la tercera se hace una llamada recursiva, utilizando la igualdad matemática *(n* + I) — (»i + !)=„ — *ni.*

cero — *m* = cero

suc(ri) — cero = suc(u)

suc(u) — suc(rrí) = *n* — *ni*

La exponenciación se basa en el producto, de la misma forma que el producto se basa en la suma, pero es importante que la distinción de casos se haga en el segundo argumento, el exponente. en vez del primero, que es la base.

exp(n.cero) = suc(cero)

exp(n. suc(/n)) = *n* * expfn. *ni)*

La especificación del predicado de igualdad necesita distinguir los cuatro casos que se obtienen al considerar las dos constructoras para cada uno de los dos argumentos. Las tres primeras ecuaciones proporcionan un resultado directamente, mientras que la cuarta es recursiva. Para la desigualdad podríamos distinguir los mismos cuatro casos, pero claramente es mucho más sencillo definir la desigualdad como la negación de la

igualdad, mediante una única ecuación.

cero == cero = cierto
cero == suc(/n) = falso
suc(n) == cero — falso
suc(n) =— suc(nr) = n == *ni*
n / ni = —■(/? - ni)

La distinción de casos para la operación < es exactamente la misma que para la resta. Mediante distinciones de casos semejantes se podrúi definir cada una de las restantes relaciones de orden de forma independiente, pero en la especificación que sigue definimos < en términos de < y 7=. y las demás relaciones en base a la simetría existente entre ellas.

cero < *111* = cierto
suc(n) < cero = falso
suc(n) < suc(m) = *11 < 111*
n < m = n < ni A *n ni*
n > ni = ni < 11
11 > 111 = 111 < 11

La distinción de casos para el máximo y el mínimo vuelve a ser la misma que para < y la resta, si bien, en este caso, las operaciones son conmutativas.

máxfcero. *ni)* = *ni*
máx(suc(/i), cero) = suc(n)
máx(suc(n). suc(tn)) = suc(máx(n. *ni))*
mínfeero. *ni)* — cero
mín(suc(/i), cero)= cero
mín(suc(n). suc(ni)) = suc(min(íi. »i))

Para las operaciones div y mod la distinción de casos no está basada en constructoras, sino en la relación de orden entre los argumentos, y por esta

razón se hace mediante condiciones eeuaeionales. En

primer lugar tenemos una ecuación de error debido a la parcialidad. En segundo lugar el caso básico es aquel en que el dividendo es más pequeño que el divisor, en cuyo caso el cociente vale cero y el resto coincide con el dividendo. Finalmente, cuando el dividendo es suficientemente grande, la tercera ecuación produce el resultado a partir de una llamada recursiva basada en la resta; intuitivamente, la división se reduce a sucesivas restas, de la misma forma que multiplicar significa realizar sucesivas sumas.

n div cero = error

n div *ni* — cero <= *n* < *m*

n div *ni* = suc((n — *m)* div *ni)* <= *m*
cero A *m* < *n*

n mod cero = error

n mod *ni* = *n* <= *n* < *ni*

n mod *ni* — *(n* — *ni)* mod *m* <= *m*
cero A *ni* < *n*

Nótese que en las condiciones de las ecuaciones anteriores *n* < *ni* es un predicado que abrevia la ecuación (zr < *m)* = cierto, según la convención explicada en la Sección 1.1.2. Lo mismo se aplica al término *m* / cero, que aplica la operación de desigualdad, y a *ni* < *n.*

La especificación de los predicados de paridad se basa en distinguir en primer lugar si el argumento es cero o un sucesor suc(zzr). Sin embargo, en el segundo caso no tenemos bastante información para obtener directamente un resultado, por lo que volvemos a

distinguir casos sobre *ni.* para ver si es cero o un sucesor. De esta forma, obtenemos en total tres casos, correspondientes a 0 (el número par más pequeño), a 1 (el número impar más pequeño) y a *n* + 2, cuya paridad es la misma que la de *n.*

es-par?(cero) = cierto
es-par?(suc(cero)) = falso
es-par?(suc(suc(z;))) = es-par?(;t)
es-impar?(cero) = falso
es-impar?(suc(cero)) = cierto
es-impar?(suc(suc(n))) = es-impar?(n)

Como ya hemos visto en casos anteriores, no es necesario definir las operaciones de forma independiente. Una posibilidad es. dada una de las dos operaciones, definir la otra como su negación, por ejemplo

es-impar?(/i) = -■es-par?^;)

Otra posibilidad es dar una definición *mutuamente recursiva* de ambas operaciones, basada en la idea de que la paridad de *n* + 1 es precisamente la contraria de la paridad de *n.*

es-par?(cero) = cierto
es-par?(suc(n)) = es-impar?(/r)
es-impar? (cero) = falso
es-impar?(suc(zi)) = es-par?(zz)

fespecificación

Especificar los *números enteros* con las siguientes operaciones:

- **cero, sucesor y predecesor,**
- **suma,**
- **producto,**
- **diferencia.**

- **cambio de signo,**
- **relación de igualdad, y**
- **relación de orden <.**

i?z.. KBE

Solución:

De la misma forma que los números naturales en el Ejercicio 1.2. los números enteros positivos se generan a partir de la constante cero aplicando sucesivas veces la operación sucesor suc que. intuitivamente. suma uno; así el término construido suc"(cero) (donde la notación suc" indica que la operación suc se aplica *n* veces) representa el número positivo *n.* De forma análoga. los números enteros negativos se generan a partir de la constante cero aplicando sucesivas veces la operación predecesor pred que. intuitivamente, resta uno; así el término construido pred"(cero) representa el número negativo — *n.* Por tanto, consideramos como constructoras las tres operaciones cero, suc y pred.

Ahora bien, los términos construidos con estas tres operaciones, además de los mencionados anteriormente, incluyen infinitos más en los cuales se aplican de forma entremezclada las operaciones suc y pred. como por ejemplo suc(pred(suc(pred(cero)))). Con la idea de que suc representa sumar uno mientras que pred representa restar uno, y teniendo en cuenta que sumar uno y restar uno se anulan mutuamente, el término anterior y cero deben ser considerados equivalentes, pues ambos representan el número 0. Dicho de otra forma, las operaciones suc y pred son mutuamente inversas, por lo que las constructoras *no son libres* y tenemos

dos ecuaciones de equivalencia entre los términos construidos.

La signatura siguiente incluye las declaraciones de todas las operaciones del enunciado, que son totales. Las relaciones de igualdad y orden son observadoras, mientras que las tres operaciones aritméticas y el cambio de signo son modificadoras.

especificación *ENTEROS*
usa *BOOLEANOS*
tipos *ent*
operaciones
cero: —>*ent* { constructora}
suc *: ent—>ent* { constructora}
pred*: ent—>ent* { constructora|
_ -I- _ *:ent ent —> ent*
_ * _*:ent ent —» ent*
_ — _*:ent ent —> ent*
cambio-signo : *ent —> ent*
_ < _ *ent ent —> boot*
_ == _ *:ent ent —► bool*
variables
n,m : ent
ecuaciones
suc(pred(n)) = *n* { inversas)
pred(sucfn)) = *n* { inversas)

El comportamiento de las operaciones modificadoras y de las observadoras se especifica mediante ecuaciones en términos de las constructoras. Para las operaciones binarias basta, en general, con distinguir los tres casos en uno de los argumentos. Como la suma y el producto son conmutativos, da igual que la distinción se haga en el primer o segundo

argumento; para variar un poco con respecto a la especificación de los números naturales en el Ejercicio 1.2. aquí consideramos las ecuaciones resultantes al distinguir casos en el segundo argumento. Las ideas intuitivas son exactamente las mismas que para las operaciones aritméticas sobre los naturales, añadiendo un tercer caso para tratar el predecesor.

n + cero = *n*

n + suc(m) = sucfti + m)

n + pred(m) = pred(zr + *ni)*

n * cero = cero

n * suc(ni) = *(n*ni)+n*

n * pred(ni) = (n*ni) —n

Con respecto a la diferencia, conviene distinguir casos sobre el segundo argumento, de forma que al restar un sucesor se convierte en un predecesor, y viceversa. Exactamente la misma idea se aplica a la operación que cambia el signo. la cual también podría ser definida en términos de la resta teniendo en cuenta la igualdad matemática — *n* = 0 — *n.*

n — cero = *n*

n — *suc(zzz)* = pred(/t — *ni)*

n — *pred(ni)* = suc(zi — *ni)*

cambio-signo(cero) = cero

cambio-signo(suc(zi)) = pred(cambio-signo(zi))

cambio-signo(pred(n)) = suc(cambio-signo(n))

Para definir la relación de orden < sobre números enteros, distinguimos los tres casos correspondientes a las tres constructoras en el primer argumento. Cuando se trata de un sucesor o predecesor, se simplifica transformando adecuadamente el segundo argumento, utilizando la idea de que un predecesor en el primer argumento se convierte en un sucesor en el segundo, y viceversa: de esta forma se llega al caso en que el primer argumento es cero. Entonces se distinguen casos en el segundo argumento según constructoras y mediante condiciones en las que tienen lugar llamadas recursivas sobre la operación <.

cero < cero = cierto

cero < pred(cero) = falso

cero < suc(zzz)= cierto <= cero < *ni*

cero < pred(zzz) = falso <= -■(cero< *ni)*

suc(zn) < *n* = *ni* < pred(n)

pred(zn) < *n* = *ni* < suc(zz)

Nótese que estas ecuaciones *no* incluyen *todos* los casos, en el sentido de que no hay ninguna ecuación que dé directamente el resultado de cero < suc(zzi) cuando -■(cero < *ni),* y lo mismo para cero < predi *ni)* cuando cero < *ni.* Sin embargo, utilizando todas las ecuaciones de la especificación (incluyendo las ecuaciones de equivalencia entre los términos construidos, que afirman que las operaciones suc y pred son mutuamente inversas) *en el orden adecuado,* sí se obtienen los resultados para todos los casos.

Finalmente, la operación de igualdad entre enteros se especifica a partir de la operación de orden

n == *ni* = *n* < *ni* A *ni* < *n*

fespecificación

A semejanza de la especificación de los números naturales (Ejercicio 1.2). a partir de == y < se pueden definir fácilmente la relación de desigualdad y las otras relaciones de orden entre enteros.

En los ejercicios que siguen usaremos los números naturales y los enteros, pero

en general los representaremos con la notación decimal habitual, en vez de utilizar la notación uñaría basada en cero y suc (y pred para los enteros).

1.4. Usando la especificación de los enteros en el Ejercicio 1.3, especificar los *números complejos* cuyas componentes en coordenadas cartesianas son números enteros, incluyendo las siguientes operaciones:

- **parte real,**
- **parte imaginaria,**
- **suma,**
- **diferencia,**
- **producto,**
- **conjugado,**
- **relación de igualdad, y**
- **valor absoluto (cuadrado del módulo).**

-------------------Solución-------------------

Ninguna de las operaciones que aparecen en el enunciado permite generar los números complejos. Como un número complejo se puede representar de forma única como un par de enteros, vamos a añadir a la signatura de la especificación una única operación constructora *libre,* que denominamos cc. que recibe dos enteros como argumentos. Entonces la parte real y la imaginaria son las proyecciones correspondientes y el resto de las operaciones se definen, de la forma habitual, en términos de las operaciones sobre las componentes enteras.

Según el tipo del resultado es inmediato distinguir las observadoras de las modificadoras en la signatura siguiente:

especificación *COMPLEJOS*
usa *BOOLEANOS. ENTEROS*

tipos *complejo*
operaciones
cc *:ent ent —> complejo* {
constructora }
re *:complejo —> ent*
im *complejo —> ent*
sumac *complejo complejo —**
complejo
restac *complejo complejo —»*
complejo
prodc *complejo complejo —>*
complejo
conj*complejo —► complejo*
_ == _ *: complejo complejo -—>*
bool
val *complejo —» ent*
variables
a. b. c. d : ent
ecuaciones
re(cc(rt. *bj) = a*
im(cc(<7. *b)) = b*
sumac(cc(«. ó). cc(c. r/)) = *CC(ÍI + c. b*
+ d)
restac(cc(o. *b),* cc(e. </)) = cc(tt — c. *b*
— d)
prodc(cc(u, /?). cele. *d})* = cc((o * c) —
*(b * d). (b * c) + (a * d))*
conj(cc(«. i>)) = cc(«. cambio-
signo(/?))
cc(a. b) == cc(c, *d) = a == b* A *C == d*
val(cc(«. *bj) = (a ** a) + *(b * b)*
fespecificación

Nótese que para la relación de igualdad sobre los complejos estamos utilizando la misma notación que para la relación de igualdad sobre los enteros, por lo que en

la ecuación que define la igualdad de complejos el símbolo == aparece *sobrecargado.* Sin embargo, el contexto permite desambiguar sin dificultad alguna: en la parte izquierda de la ecuación el símbolo denota la igualdad sobre los complejos, mientras que en la parte derecha de la misma se refiere en ambos casos a la igualdad sobre enteros. Por otra parte, aunque la ecuación val(e) = prodc(e. confie)) es matemáticamente correcta para definir el valor absoluto de un número complejo como el producto de ese número por su conjugado, no sirve en la especificación porque la función val tiene que devolver un número entero (no negativo), mientras que el término prodc(c, confie)) es un producto de complejos y es. por tanto, un término cuyo tipo es *complejc* en vez de *ent.* Una forma correcta es escribir val(e) = re(prodc(c. confie))).

1.5. Especificar las *cadenas* finitas sobre un alfabeto dado como parámetro. La especificación debe incluir las siguientes operaciones:

- crear la cadena vacía,
- añadir un elemento por la izquierda a una cadena.
- añadir un elemento por la derecha a una cadena.
- generar una cadena unitaria formada por un elemento dado.
- concatenar dos cadenas,
- calcular la longitud de una cadena,
- consultar el elemento más a la izquierda de una cadena.
- eliminare! elemento más a la izquierda de una cadena,
- consultar el elemento más a la derecha de una cadena,
- eliminar el elemento más a la derecha de una cadena,
- decidir si una cadena es vacía,
- decidir si un elemento aparece en una cadena,
- decidir si dos cadenas son iguales, y
- calcular la inversa de una cadena.

Utilizar las siguientes operaciones como constructoras:

(a) cadena vacía y añadir por la izquierda,

(b) cadena vacía y añadir por la derecha, y

(c) cadena vacía, cadena unitaria y concatenación.

Solución:

La especificación de las cadenas es

genérica o *paramétrica* con respecto a! alfabeto sobre el cual se construyen las cadenas, es decir, la especificación es independiente de los elementos concretos que constituyan el alfabeto en un uso particular de las cadenas. Por esta razón, tendremos una *especificación parametrfiada* con respecto a un parámetro que proporciona los requisitos que debe satisfacer dicho alfabeto.

Como algunas de las operaciones que debe incluir la especificación necesitan comparar elementos del alfabeto (por ejemplo para ver si un elemento aparece en una cadena o si dos cadenas son iguales), nos hace falta una operación de igualdad sobre el alfabeto. Así. el parámetro que vamos a utilizar es *ELEM=.* definido en la Sección 1.1.5, que exige como requisitos un tipo denominado *elemento* (que representa el alfabeto) y una operación de igualdad denotada == (esta notación estará sobrecargada porque vamos a denotar de la misma forma la igualdad sobre cadenas).

La siguiente signatura define la interfaz de la especificación, que es común a las tres versiones de la misma. Nótese que las cuatro operaciones para consultar y eliminar el elemento más a la izquierda y a la derecha de una cadena son parciales porque no están definidas sobre la cadena vacía.

especificación *CADENAS[ELEM=]*
usa *BOOLEANOS. NATURALES*
tipos *cadena*
operaciones
cad-vacía : *—> cadena*
añ-izq *: elemento cadena —> cadena*
añ-der *: cadena elemento —► cadena*
unit *:elemento —> cadena*
concat *: cadena cadena —> cadena*

longitud*:* *cadena* *—>* *nat*
prim-izq*:* *cadena* *—elemento*
elim-izq*:* *cadena* *—cadena*
prim-der *cadena* *—elemento*
elim-der*:* *cadena* *•—cadena*
es-cad-vacía? :*cadena* *—>* *bool*
está? *elemento cadena* *—»* *bool*
_ == _ *cadena cadena* *—>* *bool*
inversa *:* *cadena* *—>* *cadena*

variables

e, f : elemento
x. y, z : *cadena*

Apartado (a)

Cualquier cadena se puede construir de forma única a partir de la cadena vacía añadiendo sucesivamente sus elementos por la izquierda. Por tanto, cad-vacía y añ-izq constituyen un conjunto de constructoras *libres, y* no hay ecuaciones de equivalencia entre términos construidos.

Las restantes operaciones las definimos, en general, distinguiendo casos sobre las dos constructoras del tipo *cadena.* Conviene hacer notar que *no* tiene sentido distinguir casos sobre constructoras para argumentos de tipo *elemento,* puesto que este tipo "variable" viene del parámetro y no tenemos constructoras para el mismo. Cuando se utilicen las cadenas sobre un tipo concreto, el tipo *elemento* se instanciará adecuadamente. pero en la especificación genérica de las cadenas esa información

ni se tiene ni es necesaria.

La cadena unitaria se define inmediatamente en términos de las constructoras, mientras que para las operaciones añ-der y longitud distinguimos los dos casos en el argumento de tipo *cadena,* haciéndose las llamadas recursivas apropiadas en el caso no vacío. La operación de concatenación es binaria, pero no conmutativa, y es más conveniente distinguir los casos en un argumento o en el otro, según la elección de constructoras En este apartado la distinción se hace en el primer argumento, porque añadir por la izquierda tiene que ver con el "principio" de la cadena.

ecuaciones

añ-der(cad-vacía. *f)* — añ-izq(/, cad-vacía)

añ-der(añ-izq(e. x),/) = añ-izq(í>, añ-der(x./))

unit(e) = añ-izq(e. cad-vacía)

concat(cad-vacía. y) = y

concat(añ-izq(<?, x). y)= añ-izq(e. concatfx.y))

longitud(cad-vacía) = 0

longitud(añ-izq(e, x)) = 1 4-longitud(x)

Las operaciones para consultar y eliminar el elemento más a la izquierda de una cadena dan error sobre la cadena vacía y son las proyecciones en el caso no vacío, pues podríamos llamarlas las "destructoras" asociadas a la constructora añ-izq

prim-izq(cad-vacía) = error
prim-izq(añ-izq(e. x)) = *e*
ellm-izq(cad-vacía) = error
elim-izq(añ-izq(e. A^{-})) = x

En cambio, las operaciones para consultar y eliminar el elemento más a la derecha descomponen una cadena en componentes que no son los argumentos de la constructora añ-izq y, por tanto, su definición es más complicada, necesitándose distinguir más subcasos dentro del caso no vacío. Si la cadena es unitaria.

el elemento más a la derecha coincide con el más a la izquierda (es el único que hay en la cadena), y si no es unitaria, recursivamente se busca en la parte x de la derecha quitando el elemento más a la izquierda *e.* Para la operación de eliminar, la distinción de casos es la misma, pero hay que tener cuidado en la llamada recursiva, ya que si solamente escribiéramos a la derecha de la ecuación el término elim-der(x) estaríamos eliminando tanto el elemento más a la derecha de la cadena original como el de más a la izquierda *e.*

prim-der(cad-vacía) = error
prim-der(añ-izq(e. cad-vacía))= *e*
prim-der(añ-izq(e. x)) = prim-der(.v) <= -'es-cad-vacía?(x)
elim-der(cad-vacía) = error
elim-der(añ-izq(e. cad-vacía)) = cad-vacía
elim-der(añ-izq(e, x))= añ-ízq(e, elim-der(x)) <= -■es-cad-vacía?(x)

Los predicados para comprobar si una cadena es vacía y para ver si un elemento aparece en una cadena tienen una definición recursiva inmediata, utilizando la igualdad entre elementos en la definición del segundo.

es-cad-vacia?(cad-vacía) = cierto
es-cad-vacía?(añ-izq(e. x)) = falso
está? (/, cad-vacía) = falso
está?(/, añ-izqíe. x)) = *f* —= *e* v está?(/, x)

La especificación de la igualdad entre cadenas necesita distinguir los dos casos

en los dos argumentos, dando lugar a cuatro ecuaciones. Las tres primeras tienen resultados inmediatos, mientras que la cuarta realiza una llamada recursiva. Nótese la *sobrecarga* del símbolo ==: en la parte izquierda de las cuatro ecuaciones denota la igualdad sobre cadenas, mientras que en la parte derecha de la cuarta ecuación *e* == *f* compara elementos y x == y compara cadenas, dando lugar a la llamada recursiva.

cad-vacía == cad-vacía = cierto
cad-vacía == añ-izq(e, x) = falso
añ-izq(e, x) == cad-vacía = falso
añ-izq(e, x) == añ-izq(/, y) = *e* == *f* A X == y

Finalmente, para calcular la inversa se distinguen casos, realizándose una llamada recursiva en el caso no vacío, que hace uso de la simetría entre añadir por la izquierda y añadir por la derecha.

inversa(cad-vacía) — cad-vacía
inversa(añ-izq(e, x)) = añ-der(inversa(x), *e)*

fespecificación

Apartado (b)-------------------

Cualquier cadena se puede construir de forma única añadiendo sucesivamente por la derecha a partir de la cadena vacía. Por tanto, cad-vacía y añ-der también constituyen un conjunto de constructoras *libres.*

La situación es completamente simétrica a la del Apartado (a) y las ecuaciones que

siguen se basan esencialmente en las mismas ideas que las anteriores, teniendo en cuenta dicha simetría: ahora conviene distinguir casos en el segundo argumento para la concatenación, las operaciones para consultar y eliminar el elemento más a la derecha de una cadena son las "destructoras” asociadas a la constructora añ-der, la especificación de las operaciones para consultar y eliminar el elemento más a la izquierda de una cadena es algo más complicada y se basa en distinguir tres casos, e inversa utiliza la simetría entre añadir por la derecha y añadir por la izquierda.

ecuaciones

añ-izq(<?, cad-vacía) = añ-der(cad-vacía. e)

añ-izq(<?. añ-der(x, /)) = añ-der(añ-izq(e, x),/)

unit(e-) = añ-der(cad-vacía, *e)*

concat(x, cad-vacía) = x

concat(x. añ-der(y. /)) = añ-der(concat(x. y), /)

longitud (cad-vacía) = 0

longitud(añ-der(x. /)) = longitud(x) + 1

prim-izq(cad-vacía) = error

prim-izq(añ-der(cad-vacía,/)) = *f*

prim-izq(añ-der(x./)) = prim-izq(x) <= ^es-cad-vacía?(x)

elim-izq (cad-vacía) = error

elim-izq(añ-der(cad-vacía, /)) = cad-vacía

elim-izq(añ-der(x, /)) = añ-der(elim-izq(x). /)<= -■es-cad-vacía?(x)

prim-der(cad-vacía) = error

prim-der(añ-der(x. /)) = *f*
elim-der(cad-vacía) = error
elim-der(añ-der(x. /)) = x
es-cad-vacía? (cad-vacía) = cierto
es-cad-vacía?(añ-der(x, /)) = falso
está?(e, cad-vacía) = falso
está?(e, añ-der(x,/)) = *e* == *f* v
está?(e. x)
cad-vacía == cad-vacía = cierto
cad-vacía == añ-der(x. <?) = falso
añ-der(x. <?) == cad-vacía = falso
añ-der(x. *e)* == añ-der(y. /) = e == *f*
A X == y
inversa(cad-vacía) = cad-vacía
inversa(añ-der(x,/)) = añ-izq(/.
inversa(x))
(especificación

Apartado (c)---------------------
Cualquier cadena no vacía se puede construir concatenando sucesivamente cadenas unitarias, por lo que las operaciones cad-vacia. unit y concat constituyen un conjunto de constructoras válido para generar las cadenas. Sin embargo, estas constructoras *no son libres,* pues existen muchas formas de representar una misma cadena. Notemos en particular que la cadena vacía es el elemento neutro de la concatenación, por lo que, por ejemplo, el término concat(cad-vacía. x) se comporta como x para cualquier cadena x. Asimismo, la concatenación no es conmutativa, pero sí asociativa, por lo que el orden en el que se realiza una sene prefijada de

concatenaciones no importa. Obtenemos en consecuencia el siguiente conjunto de ecuaciones:

ecuaciones

concat(cad-vacía, x) = x { elemento neutro)

concat(x, cad-vacía) = x { elemento neutro)

concat(concat(x. y), ;)=concattx, concat(y..-)) (asociatividad)

Ahora las operaciones para añadir por la izquierda y por la derecha son modificadoras y se definen directamente en términos de las constructoras sin necesidad de distinguir casos.

añ-izq(e, x) = concat(unit(e),x)
añ-der(x. <?) = concat(x, unit(e))

La longitud se define distinguiendo los tres casos posibles para las constructoras. Notemos que la definición no depende del representante elegido; por ejemplo, si concat(x. y) = concat(x', y'). entonces longitud(x) + longitud(y) = longitud(x') + longitud(y') aunque tengamos longitud(x) longitud(x') y longitud(y) # longitud(y').

longitud(cad-vacía) = 0
longitud(unit(e)) = 1
longitud(concat(x, y)) = longitud(x) + longitud(y)

Las cuatro operaciones para consultar y eliminar el elemento más a la izquierda y a la derecha de una cadena dan error sobre la cadena vacía, pero ahora hay infinitos términos que representan dicha cadena, además de la constante cad-vacia. por lo que la ecuación que define la situación de error es condicional, utilizando el predicado es-cad-vacía? (definido más abajo) para que la ecuación se aplique a cualquier representación de la cadena vacía.

Las restantes ecuaciones para cada una de estas operaciones se basan en distinguir casos sobre las dos constructoras restantes, y en el caso de la concatenación distinguir a su vez subcasos para ver qué parte no es vacía. El subcaso en el cual ambas partes son vacías no aparece, porque entonces la cadena total es asimismo vacía, y esta

situación ya ha sido considerada en la ecuación de error.

prim-izq(x) = error 4= es-cad-vacía?(x)
prim-izq(unit(e)) = *e*
prim-izq(concat(x, y)) = prim-izq(x) 4= -■es-cad-vacía?(x)
prim-izq(concat(x, y)) = prim-izq(y) 4= es-cad-vacía?(x) A -■es-cad-vacía?(y)
elím-izq(x) = error 4= es-cad-vacía?(x)
elim-izq(unit(e)) = cad-vacía
elim-izq(concat(x, y)) = concat(elim-izq(x), y) 4= ^es-cad-vacía?(x)
elim-izq(concat(x, y)) = elim-izq(y) 4= es-cad-vacía?(x) A -■es-cad-vacía?(y)
prím-der(x) = error 4= es-cad-vacía?(x)
prim-der(unit(e)) = *e*
prim-der(concat(x, y)) = prim-der(y) 4= ->es-cad-vacía?(y)
prim-der(concat(x, y)) = prim-der(x) 4= es-cad-vacía[7](y) A -■es-cad-vacía?(x)
elim-der(x) = error 4= es-cad-vacía?(x)
elim-der(unít(e)) = cad-vacía
elim-der(concat(x, y)) = concat(x, elim-der(y)) 4=-■es-cad-vacía?^')
elim-der(concat(x, y)) = elim-der(x) 4= es-cad-vacía?(y) A -■es-cad-vacía?(X)

La especificación de los predicados para comprobar si una cadena es vacía y para ver si un elemento aparece en una cadena se basa en distinguir los tres casos de las constructoras y combinar adecuadamente las llamadas recursivas

sobre las partes en el caso de la concatenación, de forma análoga a como se hizo en la especificación de longitud.

es-cad-vacía? (cad-vacía) = cierto

es-cad-vacía?(unit(c)) = falso

es-cad-vacía?(concat(x, *y))* = es-cad-vacía?(x)A es-cad-vacía?(y)

está?(e, cad-vacía) = falso

está?(e, unit(/)) = *e* == *f*

está?(e, concat(x, y)) = está?(e, x) v está?(e, y)

Para comprobar la igualdad entre dos cadenas, en general no sirve la descomposición que proporcionan las constructoras porque, por ejemplo, podemos tener concat(x, y) = concat(x', y') aunque x ■/ x'

e *y* y'. En cambio sí puede usarse la descomposición dada al consultar y eliminar el elemento más la izquierda (o a la derecha). Esta descomposición solamente puede utilizarse cuando ambas cadenas n son vacías; los casos en que una de ellas es vacía se tratan mediante otras ecuaciones condicionales.

x == y = cierto <= es-cad-vacia?(x) A es-cad-vacía?(V)

x == y =falso <= -■es-cad-vacia?(x) A es-cad-vacía?(y)

x == y =falso <= es-cad-vacía?(x) A --es-cad-vacía?(y)

x —— *y* = prim-izq(x) == prim-izq(y) A elim-izq(x) == elim-izq(y)
<= ^es-cad-vacía?(x) A -■es-cad-vacía?(y)

Finalmente, la definición de inversa distingue los tres casos, de forma análoga a como se ha hech anteriormente para longitud, es-cad-vacía? y está?.

inversa(cad-vacía) = cad-vacía
inversa(unit(e)) = unit(e)
inversa (concatfx, y)) = concat(inversa(v), inversa(.r))

fespecificación

1.6. Extender cada una de las especificaciones de las cadenas sobre un alfabeto dadas en el Ejercicio 1.5 añadiendo las operaciones siguientes;

- rota-izq(<?i .. .«„) = *ai.. .a„a\,*
- rota-der(<7| .. .«„) — *a_na\ ...a_n-t.*
- i-ésimo(fli ...*a„, i) = ai,* y
- medio(«i .,.*a„)* = a(„+i)div2-

------------------Solución------------------

La extensión pedida continúa siendo una *especificación paranietrizada* con respecto al parámem *ELEM=* (aunque la operación de igualdad no sea necesaria para definir estas operaciones) y la signatur que presentamos a continuación es válida para las tres versiones de la especificación.

La operación i-ésimo es parcial porque solamente tiene sentido cuando el índice í. dado como segunda argumento, está en el rango 1 < *i* < longitud(x), para la cadena x dada como primer argumento. L operación medio también es parcial porque no está definida para la cadena

vacía.

especificación *CADENAS+[ELEM=]*

usa *CADENAS[ELEM=]*

operaciones

rota-izq *cadena —>* *cadena*

rota-der*cadena —>* *cadena*

i-ésimo : *cadena nat —* p*
elemento

medio*: cadena —> p elemento*

variables

e, f : elemento

x *: cadena*

n : nat

Apartado (a)-----------------

La descomposición de una cadena no vacía, proporcionada por la constructora añ-izq. es justament< la apropiada para la operación rota-izq que mueve el elemento más a la izquierda al extremo derecho.

ecuaciones

rota-izq(cad-vacía) = cad-vacia

rota-izq(añ-izq(í>. _v)) = añ-der(A’.e)

En cambio, para la operación rota-der necesitamos acceder al elemento más a la derecha, para moverlo al extremo izquierdo. Como la constructora añ-izq no proporciona esta descomposición, usamos las “destructoras" prim-der y elim-der. en vez de distinguir casos por constructoras.

rota-der(cad-vacía) = cad-vacía

rota-der(A’) = añ-izq(prim-der(x).
elim-der(A')) <= -'es-cad-vacía?(x)

La operación i-ésimo se define recursivamente sobre el índice dado como segundo argumento. Aparte de la

situación de error ya comentada antes, el caso básico es aquel en el que el índice vale I. y por tanto el elemento í-ésimo coincide con el más a la izquierda. El caso recursivo reduce *n* a *n* — I. quitando el elemento más a la izquierda de la cadena.

i-ésimo(A‘. *n)* = error <= *n* == 0 V *n* > longitud(A')

i-ésimo(añ-izq(e. A'). 1) = *e*

i-ésimo(añ-izq(e. A-), n) = i-ésimo(A. *n* — 1) <= *n* > 1 A *n* — 1 < longitud(A)

Otra distinción de casos equivalente es la siguiente:

i-ésimofcad-vacía. n) = error

i-ésimo(añ-izq(e, .r). 0) = error

i-ésimo(añ-izq(e. A-). 1) = *e*

i-ésimo(añ-izq(e. A), ii) = i-ésimo(x. *n* — 1) <= *n* > 1

La operación medio es un caso particular de i-ésimo, en el sentido de que

medio(A') = í-ésimo(A', (longitud(A’) + 1) div 2)

Pero también la podemos definir de forma independiente, distinguiendo los dos casos sobre constructoras y dentro del segundo los tres subcasos correspondientes a cuando la cadena tiene 1,2 o más elementos. En el último subcaso se hace una llamada recursiva, eliminando un elemento en cada extremo de la cadena original.

medio(cad-vacía) = error

medio(añ-izq(e, cad-vacía)) = *e*

medio(añ-izq(<?, añ-izq(/, cad-

vacía))) = *e*

medio(añ-izq(e. añ-izq(/. A'))) = medio(añ-izq(/. elim-der(A))) <= -■es-cad-vacíaVtA)

fespecificación

Apartado (b)

Ahora la situación es simétrica con respecto al Apartado (a). La descomposición de una cadena no vacía proporcionada por la constructora añ-der es apropiada para la operación rota-der. mientras que para la operación rota-izq usamos las “destructoras” prim-izq y elim-izq.

ecuaciones

rota-izq(cad-vacía) = cad-vacía

rota-izq(A-) = añ-der(elim-izq(A), pnm-izqfx)) <= -^es-cad-vacía?(x)

rota-der(cad-vacía) = cad-vacía

rota-der(añ-der(A–, c)) = añ-izq(e,A‘)

La ecuación de error en la definición de i-ésimo es la misma de antes, pero en esta ocasión la reclu sión es diferente. Cuando el índice, dado como segundo argumento, coincide con la longitud de la cadena

dada como primer argumento, el resultado es el elemento más a la derecha de la cadena, proporcionado directamente por la descomposición de la constructora añ-der. Si el índice es más pequeño, quitamos ese elemento en la cadena y hacemos una llamada recursiva, sin cambiar el valor del índice.

i-ésimofx, 11) = error 4= *n* == 0 v *n* > longitud(.v)

i-ésimo(añ-der(x, *e), n)* = *e* 4= *n* == longitud(.v) 4- 1

i-ésimo(añ-der(x, *e), n)* — i-ésimofx, *n)* 4= 0 < *n* A *n* < longitud(.r)

La idea para definir medio de forma independiente de i-ésimo es la misma que la del Apartado (a).

medio(cad-vacía) = error

medio(añ-der(cad-vacía, *e))* = *e*

medio(añ-der(añ-der(cad-vacía, <?), /)) = *e*

medio(añ-der(añ-der(jr, *e). f))* = medio(añ-der(elim-izq(.r). e)) 4= -■es-cad-vacía?(.r)

fespecificación

Apartado (c)

Cuando las constructoras son cad-vacía, unit y concat, la descomposición que proporcionan estas constructoras no sirve para definir ninguna de las cuatro operaciones que hay que especificar. Por esta razón, en las ecuaciones siguientes se distinguen casos mediante el predicado es-cad-vacía? y se usan las "destructoras"

prim-izq. elim-izq. prim-der o elim-der, según interese. Por supuesto, esta solución también es aplicable en los apartados anteriores, pero allí hemos preferido aprovechar las constructoras disponibles para distinguir casos y así ver diferentes formas de especificar las mismas operaciones.

ecuaciones

rota-izq(.v) = cad-vacía 4= es-cad-vacía? (.v)

rota-izq(v) = añ-der(elim-izq(x), prim-izq(.r)) s= -■es-cad-vacía?(.v)

rota-der(.v) = cad-vacía 4= es-cad-vacía?(.v)

rota-der(.v) = añ-izq(prim-der(.v), elim-der(.v)) 4= -■es-cad-vacía?(.r)

i-ésimo(.r, ti) = error 4= *n* == 0 V *n* > longitud(.v)

i-ésimo(.t, 1)   = prím-izq(.v) 4= -■es-cad-vacía?(.v)

i-ésímo(.v, n) = i-ésimo(elim-izq(.v), *n* — 1)

4= --es-cad-vacía?(,v) A 0 < *n* *A* *ti* < longítud(.v)

medio(.r) = error s= es-cad-vacía?(,v)

medio(.v) = prim-izq(.r) 4= 0 < longitud(.v) A longitud(.v) < 2

medio(.v) = medio(elim-izq(elim-der(.v))) 4= longitud(.v) > 2

fespecificación

1.7. Especificar los *conjuntos* finitos sobre un tipo de elementos dado como parámetro. La especificación debe

incluir al menos las siguientes operaciones:

- crear un conjunto vacío,
- formar un conjunto unitario con un elemento dado,
- añadir un elemento a un conjunto,
- relación de pertenencia entre un elemento y un conjunto,
- predicado para decidir si un conjunto es vacío,
- quitar un elemento a un conjunto,
- unión de conjuntos,
- intersección de conjuntos,
- diferencia de conjuntos, y
- cardinal de un conjunto.

------------------Solución---------------

Al igual que la especificación de las cadenas en el Ejercicio 1.5, la especificación de los conjuntos está *parametrizada* con respecto al parámetro *ELEM=,* definido en la Sección 1.1.5, que exige un tipo denominado *elemento* sobre el cual se construyen los conjuntos así como una operación de igualdad denotada ==. que ahora nos hace falta concretamente para especificar la relación de pertenencia y la operación que quita un elemento a un conjunto.

La siguiente signatura declara todas las operaciones como totales.

especificación *CONJUNTOS[ELEM=]*
usa *BOOLEANOS. NATURALES*
tipos *conjunto*
operaciones
cjto-vacío : —> *conjunto*

añadir : *elemento conjunto —> conjunto*
unit *:elemento —> conjunto*
está? *:elemento conjunto —> bool*
es-cjto-vacío? : *conjunto —» bool*
quitar *: elemento conjunto —> conjunto*
unión*:conjunto conjunto —> conjunto*
intersección *: conjunto conjunto—> conjunto*
diferencia *: conjunto conjunto—> conjunto*
cardinal *: conjunto —> nat*

variables

e, f : elemento
x, y, z : *conjunto*

Una primera posibilidad es considerar como constructoras cjto-vacío y añadir, pues todo conjunto/im/o puede obtenerse a partir del vacío añadiendo elementos de uno en uno. En tal caso, el orden en que se añaden los elementos no importa, y tampoco el número de veces que se añade un mismo elemento, puesto que un conjunto se caracteriza simplemente por los elementos que pertenecen a él. Por consiguiente, las constructoras *no son libres y* tenemos ecuaciones de equivalencia entre términos construidos que expresan la “conmutatividad” y la “idempotencia" de la operación añadir.

ecuaciones { primera versión)

añadir(e, añadir!/, x)) = añadir!/, añadirle, x)) { conmutatividad }

añadirle, añadirle, x)) = añadirle, x)
(idempotencia }

La operación unit se define directamente en términos de las constructoras.

unit(e) = añadirle, cjto-vacío)

Los predicados que comprueban si un conjunto es vacío y si un elemento está en un conjunto se definen de forma sencilla distinguiendo los dos casos para las constructoras.

es-cjto-vacío?fcjto-vacío) = cierto es-
cjto-vacío?lañadir(e, x)) = falso
está?fe, cjto-vacío) = falso
está?(e, añadir!/, x)) = e == *f* v
está?(e,x)

La operación quitar podría ser considerada como parcial decidiendo que solo tiene sentido quitar un elemento cuando pertenece al conjunto dado como segundo argumento. Una alternativa es considerar que quitar es total, de forma que al eliminar un elemento que no pertenece al conjunto, este se queda igual. Esta última posibilidad es la que se define en las ecuaciones que siguen. Se distinguen casos sobre

las constructoras de conjuntos y, en el caso no vacío, dos subcasos según que el elemento que se quita coincida o no con el que se añade.

quitarle, cjto-vacío) = cjto-vacío

quitarle, añadirle, x)) = quitarle, x)

quitarle, añadir!/, x)) — añadir!/, quitarle, x)) <= e *f*

La segunda ecuación en la definición anterior de quitar merece un par de comentarios. En primer lugar, nótese que la variable e aparece dos veces en el lado izquierdo, con lo que se hacen coincidir los dos elementos; otra posibilidad equivalente sería usar dos variables diferentes *e y f* como en la tercera ecuación, y tener una condición *e* == / (o también *e = /):*

quitarle, añadir!/, x)) = quitar(e.x) <= *e == f*

En ambos casos se trata de cubrir el caso contrario al cubierto por la tercera ecuación. Nótese que. de acuerdo con la convención explicada en la Sección 1.1.2, *e* == / en la condición anterior es una abreviatura de la ecuación *(e* == /) = cierto que es equivalente a la ecuación *e* = / por los requisitos impuestos sobre el predicado de igualdad en la Sección 1.1.5. Por estas razones, es equivalente utilizar la misma variable a imponer una condición de la forma anterior, ya que en cualquier caso se exige que dos elementos coincidan.

Por otra parte, uno podría esperar, erróneamente, que el lado derecho de

la segunda ecuación fuera simplemente el término x; esto es incorrecto porque las ecuaciones que aparecen en la especificación han de ser aplicables a todos los términos. Así, en este caso no podemos garantizar que en el término que representa al conjunto no haya repeticiones (en el sentido de haber añadido varias veces un mismo elemento) y, por tanto, tenemos que hacer una llamada recursiva mediante el término quitarle, x). para asegurarnos de que al final se ha quitado "por completo" el elemento *e* en el conjunto x.

Como la unión es conmutativa, se pueden distinguir casos en el primero o en el segundo argumento. Las ecuaciones siguientes consideran la primera posibilidad, pero la otra sería equivalente. Para la intersección ocurre lo mismo, pero en esta ocasión también hemos de distinguir subcasos según que el elemento que se añade esté o no en el conjunto dado como segundo argumento; para ello usamos el predicado está? en la condición.

unión(cjto-vacío, y) = y

unión(añadir(e, x), y) = añadirle, uniónfx, y))

mtersección(cjto-vacío, y) = cjto-vacío

intersección(añadir(e, x), y) = intersección(x, y) <= -■está?(e, y)

intersección(añadir(e, x), y) = añadirle, intersección(x, y)) <= está?(e, y)

La diferencia entre conjuntos no es

conmutativa, por lo que es importante ver en qué argumento se distinguen casos. Las ecuaciones siguientes lo hacen en el segundo argumento y se basan en la operación quitar definida anteriormente, teniendo en cuenta que un conjunto no varía al quitarle un elemento que no contiene.

diferencia(.v, cjto-vacío) — x

diferencialx, añadir/?, y)) =
diferencia(quitar(e, x), y)

Para calcular el cardinal de un conjunto distinguimos los casos correspondientes a las dos constructoras. pero en el caso no vacío hay que tener cuidado para no contar un mismo elemento más de una vez. Como hemos visto en las ecuaciones de la operación modificadora quitar, el resultado tiene que ser independiente del representante elegido. Al no poder garantizar que en el término que representa al conjunto no haya repeticiones, usamos quitarle, x) en el lado derecho de la ecuación para aseguramos de que el elemento *e* ya no vuelve a contarse en la llamada recursiva.

cardinal(cjto-vacío) = 0

cardinal(añadir(<’. x)) =
cardinal(quitar(e, x)) + 1

(especificación

Una segunda posibilidad es considerar como operaciones constructoras el conjunto vacío, la operación para formar conjuntos unitarios y la unión de

conjuntos. Estas constructoras tampoco son libres, y las ecuaciones de equivalencia entre términos construidos expresan las propiedades conmutativa, asociativa e idempotente de la unión de conjuntos, que además tiene al conjunto vacío como elemento neutro.

ecuaciones { segunda versión)

unión(x, y) =unión(y. x) (conmutatividad}

unión(x, uníón(y, z)) = unión(unión(x, y), z) (asociatividad)

unión(x. x) = x (idempotencia }

unión(x, cjto-vacío) = x (elemento neutro)

Ahora es sencillo definir añadir en función de las constructoras mediante una única ecuación, y definir recursivamente los predicados está? y es-cjto-vacío?. distinguiendo para cada predicado tres casos sobre las constructoras: dos básicos y uno recursivo.

añadirte, x) = unión(unit(e). x)

está?(e, cjto-vacío)= falso

está?(e, unit(/))= *e == f*

está?(e. uniónfx, y)) = está?(e,x) V está?(<?, y)

es-cjto-vacío? (cjto-vacío) = cierto

es-cjto-vacío?(unit(e))— falso

es-cjto-vacío?(unión(x. y)) = es-cjto-vacío?(x) A es-cjto-vacío?(y)

La especificación de la operación quitar distingue tres casos sobre las constructoras y. en el caso unitario, dos

subcasos; en el primero de estos, se ve que el elemento que se quita coincide con el único que hay en el conjunto, para lo que se usa la misma variable *e* dos veces en el lado izquierdo de la ecuación.

quitar(e. cjto-vacío) = cjto-vacío
quitar(e. unit(e)) = cjto-vacío
quitar(e. unit(/)) = unit(/) <= *e* / *f*
quitarte. unión(x, y)) = unión(quitar(e, x). quitarte, y))

La intersección y la diferencia se definen distinguiendo los tres casos para las constructoras. En la primera operación el argumento sobre el que se hace la distinción de casos no importa. En concreto, las ecuaciones siguientes lo hacen sobre el primero, distinguiendo dos subcasos en el caso unitario, según el elemento dado esté o no en el segundo conjunto. En la segunda operación, al no ser conmutativa, conviene tener cuidado con el argumento sobre el que se distinguen los casos, resultando más adecuado el segundo. Usamos la operación quitar en el caso unitario y realizamos dos llamadas recursivas en el caso de la unión.

intersección(cjto-vacío. y) = cjto-vacío
intersección(unit(<?). y) = cjto-vacío <= -'está?(e, y)
intersección(unit(<?), y) = unit(e) <= está?(<?, y)

intersección (unión (x, z), y) = unión (intersección (x. y). intersección(z. y))

diferencia(x, cjto-vacío) = x

diferencia(x, unit(e)) = quitarte, x)

diferencia(x, unión(y, z)) = diferencia(diferencia(x, y), z)

Finalmente, la especificación de cardinal distingue los tres casos; los dos primeros dan lugar a un resultado directo, mientras que en el caso de la unión se hacen tres llamadas recursivas. Hay que tener

en cuenta que los elementos que aparecen en la intersección de los dos conjuntos que se unen solamente tienen que contarse una vez. por lo que se calcula el cardinal de esa intersección y su valor se resta a la suma de los otros dos cardinales. El resultado de esta forma de calcular el cardinal es independiente de la forma en que se descomponga un conjunto como unión de otros dos.

cardinal (cjto-vacío) = 0
cardinal(unit(e)) = 1
cardinal(unión(.r, y)) = (cardinal(.v) + cardinal! y)) — cardinal! intersección í.v. y)) **fespecificación**

Especificar los *multiconjuntos* finitos sobre un tipo de elementos dado como parámetro. La **especifica**ción debe incluir las siguientes operaciones:

- crear el multiconjunto vacío.
- añadir un elemento,
- calcular la multiplicidad de un elemento.
- predicado para decidir si un multiconjunto es vacío.
- quitar una aparición de un elemento,
- borrar (todas las apariciones de) un elemento,
- unión de multiconjuntos,
- intersección de multiconjuntos,
- diferencia de multiconjuntos,
- calcular el cardinal, teniendo en cuenta las multiplicidades de los elementos, y
- calcular el cardinal, contando

solamente los elementos distintos.

Solución:

Las principales ideas para la especificación de este tipo de datos siguen el mismo patrón que las vistas en el Ejercicio 1.7 sobre conjuntos, y más concretamente en la primera versión de las ecuaciones.

En primer lugar, la especificación de los multiconjuntos está *parametrízada* con respecto al parámetro *ELEM=.* que requiere un tipo denominado *elemento* junto con una operación de igualdad ==.

En segundo lugar, como vemos en la siguiente signatura, todas las operaciones son totales. En particular. las operaciones quitar y borrar se consideran totales, haciendo que eliminar un elemento que no pertenece a un multiconjunto deje igual al multiconjunto.

especificación *MULTICONJUNTOS[ELEM=]* **usa** *BOOLEANOS, NATURALES* **tipos** *multiconjunto* **operaciones**

mcjto-vacío : —* *multiconjunto* {constructora]

añadir *: elemento multiconjunto —> multiconjunto* (constructora }

multip *elemento multiconjunto* -—> *nat*

es-mcjto-vacío? *multiconjunto* —» *bool*

quitar *:elemento multiconjunto* —-> *multiconjunto*

borrar *elemento multiconjunto* —>

multiconjunto

unión *multiconjunto multiconjunto* —»
multiconjunto

intersección : *multiconjunto*
multiconjunto —> *multiconjunto*

diferencia*multiconjunto multiconjunto*
—■>*multiconjunto*

cardinal : *multiconjunto* —>
nat

cardinal-dist : *multiconjunto* —* *nat*

variables

e. f: elemento

x. y : multiconjunto

Consideramos como operaciones constructoras el multiconjunto vacío, mcjto-vacío. y añadir, puesto que todo multiconjuntoJiníro puede ser construido a partir del vacío, añadiendo sucesivamente elementos de uno en uno. En este caso, las constructoras tampoco son libres, pues hay una ecuación de equivalencia entre términos construidos, debido a que no importa el orden en que se añaden los elementos (conmu-tatividad). En cambio, ahora sí importa el número de veces que se añade un mismo elemento, pues ese número es precisamente su multiplicidad en el multiconjunto.

ecuaciones

añadirte, añadir(/,x)) = añadirf/, añadirte, x)) { conmutatividad }

Para definir el resto de las operaciones (observadoras y modificadoras) en términos de las constructoras conviene, en general, distinguir los dos casos y, en el caso no vacío, si hace falta, comparar con el otro argumento, que es un elemento.

es-mcjto-vacío? (mcjto-vacío) = cierto

es-mcjto-vacío?(añadir(e. x)) = falso

multipfe, mcjto-vacío) = 0

múltiple. añadirte, x)) = multip(e, x) + 1

multipte, añadir(/, x)) = multipte, x) 4= *e f*

Nótese que las especificaciones de las operaciones quitar y borrar son muy similares. La única diferencia aparece en la segunda ecuación, en la cual para quitar se devuelve directamente el término x, pues solamente hay que quitar una aparición del elemento, mientras que para borrar se hace una llamada recursiva borrarte, x), porque hay que quitar *todas* las apariciones del elemento en el multiconjunto.

quitarte. mcjto-vacío) = mcjto-vacío
quitarte, añadirte, x)) = x
quitarle, añadirt/, x)) = añadir(/, quitarte, x)) 4= e / /
borrarte, mcjto-vacío) = mcjto-vacío
borrarte, añadirte, x)) = borrarte, x)
borrarte, añadirf/, x)) = añadir(/, borrarte, x)) <= *e / f*

Las siguientes definiciones de las operaciones unión, intersección y diferencia para multiconjuntos siguen exactamente el mismo patrón que las vistas para las operaciones con el mismo nombre sobre conjuntos en la primera versión de la solución del Ejercicio 1.7. La única diferencia es que las condiciones para la intersección se basan en la operación multip, que calcula la multiplicidad, en vez de en el predicado de pertenencia, el cual no es necesario en esta especificación dado que un elemento está en un multiconjunto si y solo si su multiplicidad es mayor que cero.

unión(mcjto-vacío, y) = *y*
unión(añadir(e. x), y) = añadirte,

unión(x. y))

intersección(mcjto-vacío, y) = mcjto-vacío

intersección(añadir(e, x), y) = intersección(x, y) 4= multipte, y) == 0

intersección(añadirte, x), y) = añadirte, intersección(x, quitarte, y))) 4= multipte, y) > 0

diferenciatx, mcjto-vacío) = x

diferencíate, añadirte, y)) = diferencia(quitar(e, x), y)

Para terminar, la diferencia entre las dos operaciones que calculan el “cardinal" de un multiconjunto aparece en el argumento de la llamada recursiva. En la primera operación, cardinal, como se tienen en

cuenta las multiplicidades, la llamada es cardinal(x). de forma que se volverá a contar el elemento *e* si vuelve a aparecer, mientras que en la segunda operación, cardinal-dist. que solo cuenta los elementos distintos y no tiene en cuenta sus multiplicidades, la llamada es cardinal-dist(borrar(e. x)) que garantiza que el elemento *e* ya no se volverá a contar.

cardinal(mcjto-vacío) = 0
cardinal(añadir(e, x)) = cardinal(x) + 1
cardinal-dist(mcjto-vacío) = 0
cardinal-dist(añadir(e, x)) — cardinal-dist(borrar(e. x)) + 1

fespecificación

Especificar los *polinomios con coeficientes naturales* **en una indeterminada, con las operaciones si**guientes:

- polinomio nulo,
- sumar un monomio a un polinomio,
- suma de dos polinomios,
- producto,
- calcular el coeficiente de cierto grado,
- predicado para decidir si un polinomio es nulo,
- calcular el grado, y
- evaluar un polinomio para un valor dado de la indeterminada.

Solución:

Empezamos escribiendo la signatura que declara como totales todas las

operaciones del enunciado, junto con una operación *privada* mult-mono que multiplica un monomio por un polinomio, cuya utilidad se verá más adelante.

especificación *POLINOMIOS-NATURALES*
usa *BOOLEANOS. NATURALES*
tipos *polinomio*
operaciones
poli-nulo : —> *polinomio* (constructora }
sumar-mono*nat nat polinomio* —> *polinomio*{ constructora)
sumar : *polinomio polinomio* —> *polinomio*
multiplicar*polinomio polinomio* —► *polinomio*
coeficiente *:nat polinomio* ■—> *nat*
es-poli-nulo?*polinomio* —> *bool*
grado*:* *polinomio* —> *nat*
eval *: polinomio nat* —* *nat*
operaciones privadas
mult-mono *nat nat polinomio* —► *polinomio* **variables**
c, *d, i. j. n : nat p,q : polinomio*

Todo polinomio (sobre naturales) no nulo se puede representar de forma única como una suma de la forma
COA'O + C|X^{l} + C2X‟ + • • • + c_n.v^{n}.
donde los *ci* son coeficientes naturales, *c„* / 0. y *n* es el grado del polinomio. Naturalmente, podemos eliminar de la suma aquellos monomios cuyo coeficiente sea 0. Utilizando las operaciones de la especificación, tal polinomio se puede representar de la forma

sumar-mono(co, 0, sumar-mono(ci, 1 ....sumar-mono(c„. *n.* poli-nulo)...)).

Por tanto, las operaciones poli-nulo y sumar-mono sirven como constructoras. Necesitamos ecuaciones de equivalencia entre términos construidos, puesto que sumar un monomio con coeficiente cero no modifica el polinomio, la suma de monomios es conmutativa, y sumar dos monomios con el mismo exponente es equivalente a sumar un único monomio con la suma de los coeficientes y el mismo exponente.

ecuaciones

sumar-mono(0. *i. p) = p*

sumar-mono(c, *i.* sumar-mono(r/. *j, pY)* = sumar-mono(rf, *j.* sumar-mono(c, *i.* p))

sumar-mono(c, *i.* sumar-mono(r/, *i, pY)* = sumar-mono(c + *d. i. p)*

La especificación de la suma distingue dos casos para las constructoras en el primer argumento (debido a la conmutatividad de la suma sería equivalente hacer la distinción sobre el segundo argumento), devolviendo el segundo argumento al sumar el polinomio nulo, que es el elemento neutro de la suma, y haciendo una llamada recursiva cuando se suma un monomio.

sumar(poli-nulo. *q)* = *q*

sumar(sumar-mono(c. *i. pY q)* = sumar-mono(c. *i.* sumar(p, *q))*

Antes de especificare! producto de polinomios, especificamos la operación privada que multiplica un monomio por

un polinomio, distinguiendo casos en el tercer argumento, que es de tipo *polinomio.* Cuando el polinomio es nulo, el resultado también lo es; mientras que si es una suma de un monomio, se aplica la distributividad del producto con respecto a la suma, para hacer por un lado una llamada recursiva con el polinomio restante *q* y por otro un producto de dos monomios, para lo cual se multiplican sus coeficientes y se suman sus exponentes, obteniéndose así un monomio que se suma al resultado de la llamada recursiva.

mult-mono(c, *i.* poli-nulo) = poli-nulo

mult-mono(c, *i,* sumar-mono(<7, *j, q))* = sumar-mono(c * *d, i + j.* mult-mono(c, *i. qY)*

Para multiplicar polinomios, basta distinguir casos en el primer argumento (en el segundo sería equivalente por la conmutatividad). Cuando el polinomio es nulo, el resultado también lo es; mientras que si es una suma de un monomio, se aplica la distributividad del producto con respecto a la suma para hacer por un lado una llamada recursiva con el polinomio restante *q.* y por otro, el producto de un monomio por un polinomio, para lo cual se utiliza la operación auxiliar mult-mono que acabamos de especificar.

multiplicar(poli-nulo, *q)*— poli-nulo

multiplicar(sumar-mono(c. *i. p), q) =* sumar(mult-mono(c, *i. q),* multiplicar! p. r/))

Para calcular el coeficiente asociado a un exponente dado, distinguimos casos sobre constructoras y cuando se trata de la suma de un monomio comparamos los correspondientes exponentes; el subcaso en que coinciden se especifica usando una misma variable *i* dos veces en el lado izquierdo de la segunda ecuación.

coeficiente^'. poli-nulo) = 0

coeficiente^, sumar-monote, *i,* p)) = c + coeficiente (/'. *p)*

coeficiente(í. sumar-mono(c. *j,* p)) = coeficiente^'. p) <= *i j*

El predicado para comprobar si un polinomio es nulo distingue los dos casos sobre constructoras y hace una llamada recursiva en el segundo caso, pues el polinomio nulo también puede representarse como una suma de monomios nulos (con coeficiente 0).

es-poli-nulo?(poli-nulo) = cierto

es-poli-nulo?(sumar-mono(c. *i. p))* = *c* —= 0 A es-poli-nulo?(p)

Para calcular el grado distinguimos casos sobre las constructoras del polinomio: en el segundo caso, cuando se suma un monomio nulo se descarta y se hace una llamada recursiva, mientras que cuando el monomio no es nulo (c / 0), teniendo en cuenta que los coeficientes son números naturales que no pueden anularse entre sí al sumar, el grado será el máximo (operación disponible sobre naturales, según la especificación del Ejercicio 1.2) entre el grado del monomio (que es z) y el

grado del polinomio restante *p,* que se calcula recursivamente.

grado(poli-nulo) = 0

grado(sumar-mono(0, *i. p)) =* grado(p)

grado(sumar-mono(c. *i, p))* — máx(í, grado(p)) <= *c r* 0

Para evaluar un polinomio sobre un valor *n* distinguimos casos sobre las constructoras del polinomio: en el caso nulo el resultado es 0, mientras que en el caso de la suma de un monomio se hace una llamada recursiva y se suma el valor obtenido al evaluar el monomio sobre *n,* lo cual se consigue utilizando las operaciones apropiadas sobre los números naturales (exponenciación, producto y suma).

eval(poli-nulo. zz) = 0

eval(sumar-mono(c, *i. p), zz) = c ** exp(n. z) + eval(p. zz) **fespecificación**

Modificar adecuadamente la especificación de los polinomios en la solución del Ejercicio **1.9 para que** los coeficientes sean números *enteros* en vez de naturales.

Solución:

Si los coeficientes son números *enteros* en vez de naturales, las únicas operaciones cuya especificación hay que cambiar son es-poli-nulo? y grado, debido a que un coeficiente positivo puede ser anulado al sumar uno negativo (con el mismo exponente). Usamos una operación auxiliar quitar-exp. declarada

como *privada* en la signatura que sigue, que "anula" el coeficiente correspondiente a un exponente dado.

especificación *POLINOMIOS-ENTEROS*
usa *BOOLEANOS. ENTEROS. NATURALES*
tipos *polinomio*
operaciones
poli-nulo —> *polinomio* { constructora }
sumar-mono *: ent nat polinomio —> polinomio* {constructora}
sumar *polinomio polinomio —> polinomio*
multiplicar *: polinomio polinomio* -—► *polinomio*
coeficiente *: nat polinomio* —*■ *ent*
es-poli-nulo?*polinomio* —>■ *bool*
grado*polinomio* —> *nat*
eval *: polinomio ent* —> *ent*
operaciones privadas
mult-mono *ent nat polinomio* —> *polinomio*
quitar-exp : *nat polinomio* —> *polinomio*
variables
c. *d. n : ent*
i, j : nat
p, q : polinomio

A continuación solamente mostramos las ecuaciones que son nuevas o necesitan ser modificadas con respecto a la especificación *POLINOMIOS-NATURALES* de los polinomios con coeficientes naturales vista en el Ejercicio 1.9.

Primero especificamos la operación

auxiliar quitar-exp, distinguiendo los dos casos de las constructoras de polinomios y dos subcasos en el caso de la suma de un monomio, según coincida el exponente que se quita con el que se suma (nótese la aparición de la misma variable *i* dos veces en el lado izquierdo de la segunda ecuación) o no (lo cual se comprueba con la condición *i j).* En ambos subcasos se hace una llamada recursiva para asegurarse de quitar posibles apariciones repetidas del mismo exponente.

ecuaciones

quitar-exp(í, poli-nulo) = poli-nulo

quitar-exp(í, sumar-mono(c, *i,* p)) = quitar-exp(í, *p)*

quitar-exp(í, sumar-mono(c. *j. p))* = sumar-mono(c. *j.* quitar-exp(z, p)) <= *i j*

El caso recursivo del predicado es-poli-nulo? es ahora más complejo debido, como decíamos más arriba, a que ciertas sumas de monomios pueden anularse entre sí. Por ello, consideramos el coeficiente “global" asociado al exponente i, que se obtiene mediante el término *c* + coeficiente^, *p),* y vemos si es cero; el polinomio será nulo cuando el resto de los coeficientes sean también cero, lo cual se comprueba con una llamada recursiva, asegurándose antes de que el exponente *i* ya no se vuelva a tratar, usando para ello la función auxiliar quitar-exp.

es-poli-nulo?(poli-nulo) = cierto

es-poli-nulo?(sumar-mono(c. i, p)) = *c* + coeficiente(í, *p)* == 0 A es-poli-

nulo?(quitar-exp(í. p);

La especificación de grado sigue la misma idea que la de es-poli-nulo? en el caso recursivo. Se trata por un lado el coeficiente “global” asociado al exponente *i* y por otro el resto del polinomio, una vez anulado ese exponente (nótese el uso reiterado de la operación auxiliar quitar-exp). El grado será el máximo entre *i* y el grado de quitar-exp(í, p), que se calcula recursivamente. En vez de utilizar la operación máx como se ha hecho en el caso de polinomios sobre naturales, a continuación se distinguen casos comparando ambos valores, para que se vean diferentes posibilidades.

grado(poli-nulo) = 0
grado(sumar-mono(c, *i,* p)) = *i* <= *c* + coeficiente(z. p) 0 A grado(quitar-exp(z. p)) <*i* grado(sumar-mono(c, *i. p))* = grado(quitar-exp(í, p)) <= *c* + coeficiente^, p) == 0 v grado(quitar-exp(í, p)) > *i*

fespecificación

Especificar un tipo de datos para representar *figuras* como sucesiones de *trazos* A (arriba), B (abajo), D (derecha), I (izquierda) de longitud unitaria, con al menos las siguientes operaciones:

- girar la figura 90 grados en sentido horario,
- duplicar el tamaño de la figura,
- calcular la altura máxima de la figura,
- calcular la anchura máxima de la figura, y
- calcular el área del rectángulo mínimo que contiene la figura.

--------------------Solución------------------

Especificamos primero un tipo básico de trazos que será utilizado para formar las figuras, y sobre el cual definimos la operación de girar un trazo 90 grados en sentido horario, diciendo directamente cuál es el resultado del giro para cada uno de los cuatro trazos posibles; por ejemplo, al girar 90 grados en sentido horario el trazo A (arriba) se obtiene el trazo D (derecha).

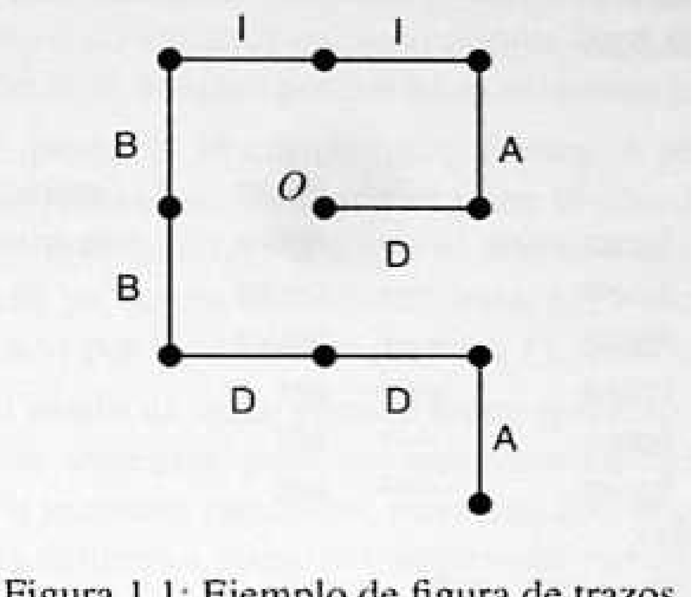


Figura 1.1: Ejemplo de figura de trazos.

especificación *TRAZOS* **tipos** *trazo*
operaciones

A —> *trazo* (constructora)
B : —**trazo* {constructora)
D : —>*trazo* (constructora)
I : —>*trazo* (constructora)
girar-trazo : *trazo* —► *trazo*

ecuaciones

girar-trazo(A) = D
girar-trazo(B) = I
girar-trazo(D) = B
girar-trazo(l) = A

fespecificación

Las figuras se crean a partir de la figura vacía añadiendo trazos, por lo que a las del enunciado añadiremos como operaciones constructoras del tipo *figura* dos operaciones fig-vacía y añ-trazo. Por ejemplo, el dibujo en la Figura 1.1 se representa mediante el término

F = añ-trazo(añ-trazo(añ-trazo(añ-trazo(añ-trazo(añ-trazo(añ-trazo(
añ-trazo(añ-trazo(fig-vacía.
D), A). I). 1). B). B). DI. D).
A)

Además de las constructoras y las operaciones del enunciado, declaramos varias operaciones privadas. El cálculo de la altura o anchura de una figura es relativamente complicado y vamos a necesitar varias operaciones auxiliares para ello. En primer lugar, para una figura dada consideramos un sistema de coordenadas enteras *local,* donde el origen se sitúa en el punto de partida de la figura (indicado como *O* en la Figura 1.1) y las operaciones coordx y

coordy devuelven las coordenadas horizontal y vertical, respectivamente. del último punto de la figura, siempre con respecto a ese origen. Por ejemplo. coordxtF) = l y coordyf *F)* = — 2.

La altura de una figura se calcula como la suma de su coordenada vertical máxima más su coordenada vertical mínima cambiada de signo (pues será menor o igual que cero): estos valores se calculan respectivamente mediante las operaciones máx y mín. De la misma forma, la anchura se calcula como la suma de la coordenada horizontal máxima (der) y de la coordenada horizontal mínima cambiada de signo (izq).

Como las coordenadas son números enteros mientras que las distancias son naturales, conviene identificar los números naturales con los enteros no negativos para simplificar la presentación de operaciones o relaciones aritméticas. Por esta razón, en la siguiente especificación utilizamos el tipo *ent* para el resultado de operaciones como altura o anchura, a pesar de que sabemos que miden distancias y. por tanto, su resultado no puede ser negativo.

especificación *FIGURAS*
usa *ENTEROS. TRAZOS*
tipos *figura*
operaciones
fig-vacía : —> *figura* {

constructora)

añ-trazo : *figura trazo —> figura*

(constructora)

girar-figura *: figura —> figura*

duplicar *: figura —> figura*

altura*: figura —> ent*

anchura *: figura —> ent*

área *: figura —> ent*

operaciones privadas

coordx: *figura —> ent*

coordy: *figura —> ent*

máx *: figura —> ent*

mín*:figura —> ent*

izq*:figura —> ent*

der *:figura —> ent*

variables

t : trazo

f : figura

Como el orden de los trazos y sus repeticiones son esenciales para definir la figura, las dos constructoras son *libres.* El resto de las operaciones se definen distinguiendo casos según las constructoras.

Para girar una figura basta girar cada uno de los trazos que la forman, y para duplicarla basta considerar cada trazo y añadirlo dos veces seguidas.

ecuaciones

girar-figura(fig-vacía) = fig-vacia

girar-figurafañ-trazof/, r)) = añ-trazo(girar-figura(/). girar-trazo(O)

duplicar(fig-vacia) = fig-vacía

duplicarfañ-trazof/, f)) = añ-trazo(añ-trazo(duplicar(/),/), r)

El origen del sistema de coordenadas

se sitúa en el punto de partida de la figura, es decir, cuando esta es vacía ambas coordenadas valen cero. En el caso no vacío, tenemos varias ecuaciones recursivas en las que distinguimos el trazo que se añade. Los trazos verticales A (arriba) y B (abajo) no modifican la coordenada horizontal, mientras que los trazos horizontales D (derecha) e I (izquierda) la modifican sumando uno y restando uno. respectivamente. Las mismas ideas se aplican a las coordenadas verticales.

coordx(fig-vacía) = 0
coordxfañ-trazof/, A)) = coordxf/)
coordxfañ-trazof/, B)) = coordxf/)
coordxfañ-trazof/. D)) = coordxf/) + 1
coordxfañ-trazof/. I)) = coordxf/) — 1
coordyffig-vacía) = 0
coordyfañ-trazof/. D)) = coordyt/)
coordyfañ-trazof/, I)) = coordyf/)
coordyfañ-trazof/, A)) = coordyf/) + 1
coordy(añ-trazo(/, B)) = coordyf/) — 1

La altura de una figura / se calcula como la suma de su coordenada vertical máxima (máxf/)) más su coordenada vertical mínima cambiada de signo (mínf/)). Estas dos operaciones auxiliares también dan cero en el caso vacío y se definen de manera recursiva distinguiento el

trazo que se añade en el caso no vacío. Los trazos horizontales no cambian las coordenadas verticales y. por tanto, no afectan al máximo ni al mínimo, el trazo vertical B (abajo) no afecta al máximo porque hace descender la coordenada vertical, y el trazo vertical A (arriba) no afecta al mínimo porque hace aumentar la coordenada vertical.

El caso interesante es el que queda: al añadir un trazo vertical A puede que la coordenada vertical se incremente (esto ocurre cuando justo antes de añadir el trazo la coordenada donde se estaba ya era el máximo, es decir, cuando máx(/) == coordy(/)) o que el máximo se mantenga porque la coordenada en la que se estaba antes de añadir ya no era el máximo (máx(/) > coordyl/)). Notemos que el caso máx(/) < coordy(/) no puede darse por la definición de máx(/). que es la coordenada vertical máxima.

La modificación que ocurre al añadir un trazo vertical B con respecto a la coordenada vertical mínima se explica de forma completamente simétrica, pero hay que tener cuidado con el signo porque las coordenadas verticales hacia abajo son números negativos, mientras que mín(/) es una cantidad no negativa que mide una distancia (por eso la definimos como la coordenada vertical mínima cambiada de signo).

altura(/) = máx(/) + mín(/) máx(fig-vacía) = 0 máx(añ-trazo(/. D)) = máx(/) máx(añ-trazo(/, I)) = *máx(f)* máx(añ-trazo(/. B)) = máx(/) máx(añ-trazo(*f.* A)) = máx(/) <= máx(/) > coordyf/) máx(añ-trazo(/. A)) = máx(/) + 1 <= máx(/) == coordy(/) mín(fig-vacía) = 0 mín(añ-trazo(/. D)) = mín(/) mín(añ-trazo(/, I)) = mín(/)

mín(añ-trazo(/, A)) = mín(/) mín(añ-trazo(/. B)) = min(/) <= mín(/) > cambio-signo(coordy(/)) mín(añ-trazo(/, B)) = mín(/) + 1 <= mín(/) == cambio-signo(coordy(/))

Teniendo en cuenta la clase de giro (90 grados a la derecha) que se ha definido sobre las figuras, una forma muy simple de calcular la anchura en términos de la altura (o viceversa) es

anchura(/) = altura(girar-figura(/))

Si queremos especificar la anchura de forma independiente de la altura, podemos aplicar las mismas ideas, utilizando operaciones auxiliares para calcular la coordenada horizontal máxima (der) y la coordenada horizontal mínima cambiada de signo (izq) en vez de máx y mín que hacen lo mismo verticalmente.

anchura!/) = izq(/) + der(/)

izq(fig-vacía) = 0 izq(añ-trazo(/. A)) = izq(/) izq(añ-trazo(/. B)) = izq(/) izq(añ-trazo(/. D)) = izq(/) izq(añ-trazo(/, I)) = izq(/) <= izq(/) > cambio-signo(coordx(/))

izq(añ-trazo(/, I)) = izq(/) + I <=
izq(/) == cambio-signo(coordx(/))
der(fíg-vacía) = 0 der(añ-trazo(/. A))
= der(/) der(añ-trazo(/. B)) = der(/)
der(añ-trazo(/. I))= der(/)
der(añ-trazo(/. D)) = der(/) <=
der(/) > coordx(/) der(añ-trazo(/, D))
= der(/) + 1 <t= der(/) == coordx(/)

Una vez definidas la altura y la anchura de una figura, es sencillo definir su área.

área(/) = altura(/) * anchura//')

(especificación

Capítulo 2

2. IMPLEMENTACIÓN DE TADs

2.1. INTRO

Habiendo dedicado el capítulo anterior a la especificación algebraica de tipos abstractos de datos, vamos a abordar en este capítulo el aspecto complementario, que es la IMPLEMENTACIÓN de los tipos de datos.

En primer lugar, las implementacioncs que vamos a realizar en este libro seguirán el paradigma de programación imperativo; esto mismo se aplica a todos los algoritmos que se van a presentar a lo largo de todo el libro.

De la misma forma que en las especificaciones de tipos de datos usamos un lenguaje abstracto de especificación, descrito en la Sección 1.1.3, para las implementaciones y algoritmos usaremos un *lenguaje abstracto de programación imperativa,* que describiremos en la siguiente sección, y que puede considerarse una simplificación de lenguajes al estilo de Pascal.

En la primera fase de la implementación de un tipo de datos hay que decidir cómo representar el tipo que se define en términos de tipos básicos, como *bool* (valores booleanos). *nat* (números

naturales), *ent* (números enteros), etc., construcciones básicas como vectores y registros, y otros tipos implementados anteriormente. De esta forma se obtiene una representación intema concreta, que se conoce en la imple- mentación pero que se oculta (por la propia metodología, independientemente de mecanismos concretos que un lenguaje pueda proporcionar además para ello) a efectos de usar el tipo de datos, que solamente se puede hacer mediante las operaciones de la interfaz del tipo de datos en cuestión.

En la segunda fase hay que escribir el algoritmo correspondiente a cada operación de la interfaz del tipo de datos y tal vez alguna operación auxiliar adicional, usando la representación concreta definida en la primera fase.

Vamos a clasificar las posibles representaciones en dos grandes clases:

- ***estáticas***, definidas típicamente en términos de un *vector* (de una o más dimensiones), de forma que el tamaño del vector está determinado por cierta constante N, es decir, la cantidad de memoria asignada a la estructura no cambia a lo largo de su vida. Como la constante N está determinada *a priori.* puede ocurrir que sea demasiado grande y se desperdicie memoria, o que la cantidad de memoria asignada se agote y esto dé lugar a errores por falta de espacio.
- ***dinámicas***, definidas típicamente en

términos de *memoria dinámica* o *punteros* (véanse la Sección 2.1.2 más adelante y el Ejercicio 2.8 al final del tema), de forma que la memoria se va asignando según se necesita, con lo cual se solucionan los problemas que se planteaban en una implementación estática, aunque tal vez a cambio de una mayor complejidad en la programación.

Una parte esencial de la programación de cualquier algoritmo, incluyendo los que implementan las operaciones de un tipo de datos, es el estudio de su coste en tiempo y en memoria. En general, cuando se habla de coste sin más nos referimos a *coste en tiempo en el caso peor.* Cuando se trate del coste en tiempo en el caso mejor, de coste en tiempo en el caso medio o de coste en espacio, lo haremos explícito.

En los algoritmos que siguen, cuando el cálculo del coste es inmediato, lo escribimos al lado del algoritmo. sin ninguna explicación adicional al respecto. Utilizamos la notación asintótica habitual, descrita por ejemplo en [Peñ98. Capítulo 1] y en casi todos los textos sobre algoritmos.

Un detalle importante es que en este libro no vamos a considerar la verificación de las implementaciones con respecto a la especificación de un tipo de datos. Esto no significa en absoluto que este tema no merezca atención, sino que se trata de un tema importante al cual se le dedica

todavía mucho trabajo de investigación, pero que nos parece que requiere demasiado formalismo a un nivel de detalle que no nos ha parecido conveniente incluir en el libro. Por tanto, confiamos en que las intuiciones que acompañan cada especificación y su correspondiente implementación sean suficientes para relacionarlas.

2.1.1. Lenguaje abstracto de programación imperativa

A continuación describimos las instrucciones que usaremos en el lenguaje abstracto de programación imperativa a lo largo del libro.

- **nada** es la instrucción que no hace nada, como su nombre indica.
- ***asignación***: La instrucción ***x := E*** asigna el valor de la expresión E a la variable x, que tiene que ser del mismo tipo que *E.* La asignación puede ser *múltiple,* de la forma $\langle x_1,\dots,x_n\rangle$ **:=** $\langle E_1,\dots,E_n\rangle$ para $n>1$, cuya ejecución asigna el valor de cada *Ei* simultáneamente a cada variable x_i, para *i* entre 1 y *n.* Cada variable debe tener el mismo tipo que el de la expresión correspondiente cuyo valor se le va a asignar, y todas las variables de la parte izquierda de la asignación múltiple tienen que ser diferentes.
- ***secuencia***: Dados dos programas P_1 y P_2, en una instrucción secuencial

  $P_1;\ P_2$

  se ejecuta primero P_1 *y,* suponiendo

que este acabe, luego se ejecuta P_2. Normalmente no escribimos el signo de secuencia al final de una línea de código.

- ***distinción de casos***: Una instrucción de la forma

casos
$B_1 \rightarrow P_1$
 $B_2 \rightarrow P2$
…
 $B_n \rightarrow P_n$
fcasos

sirve para distinguir casos de acuerdo con el valor de las expresiones booleanas B_1,…B_n conocidas como *guardas.* Se evalúan todas las guardas B_i y a continuación se ejecuta el programa *Pi* tal que Bi sea cierta. Suponemos que todas las guardas son disjuntas y que cubren todas las posibilidades.

- ***condicional***: La instrucción

si *B* **entonces** P_1 **si no** P_2 fsi

es equivalente a la siguiente distinción de casos:

casos
$B \rightarrow P_1$
 $\neg B \rightarrow P_2$
fcasos

Cuando P_2 *es* la instrucción **nada,** simplemente escribimos:

si *B* **entonces** P_1 fsi

- ***bucle***: La instrucción

mientras *B* **hacer** *P* fmientras

evalúa la condición booleana *B* y si es falsa se termina la ejecución del

bucle, mientras que si es cierta se ejecuta *P* y. a continuación, se vuelve a evaluar *B y* se repite el proceso. Esta instrucción introduce la posibilidad de no terminación en los programas, por lo que hay que asegurar su terminación mediante alguna función de cota cuyo valor se decrementa en cada iteración del bucle.

- ***bucle con contador***: La instrucción

  para *i* = *inicial* **hasta** *final* **paso** *p*
  hacer *P* fpara

  donde *inicial, final* y *p* son expresiones del mismo tipo numérico (de hecho, *p* suele ser una constante). con $p \neq 0$, es equivalente a un programa con un bucle.

  Si ***p*>0** el programa equivalente es

  i := inicial;
  mientras *i* ≤ *final* **hacer**
  P;
  i := i+p;
  fmientras

  Si ***p* < 0,** el programa equivalente es

  i := inicial;
  mientras *i* ≥ *final* **hacer**
  P;
  i := i+p;
  fmientras

  En ambos casos, *i* es una variable del mismo tipo numérico que no se modifica en *P.* Nótese que cuando el rango del contador entre *inicial y final* es vacío, no se realiza ninguna iteración del bucle. Cuando *p* = 1, que

es el caso más habitual, omitimos la mención del paso en el bucle.

- ***entrada***: la instrucción leer(c) lee un carácter del dispositivo de entrada apropiado (teclado, fichero, etc.).
- ***salida***: la instrucción imprimir(mensaje) imprime un mensaje (cadena de caracteres) en el dispositivo de salida apropiado (pantalla, fichero, impresora, etc.).
- ***error***: la instrucción `error`(mensaje) aborta la ejecución del programa e imprime un mensaje de error.
- ***comentarios***: se escriben entre llaves {esto es un comentario} en el lugar del programa que convenga.

El lenguaje abstracto está tipado, por lo que las expresiones tienen que tener el tipo apropiado para ser correctas, como ya se ha mencionado al hablar de la instrucción de asignación. En particular, todas las variables tienen un tipo asociado, como veremos a continuación. Tenemos los siguientes tipos y construcciones de *tipos básicos:*

- ***bool***, con valores booleanos cierto y falso, y las operaciones booleanas habituales de negación conjunción $\land$ y disyunción $\lor$. En algunas ocasiones es importante utilizar las versiones de estas dos últimas operaciones que se evalúan *con cortocircuito:* $\land_c$, que cuando el valor de su primer argumento es falso ya devuelve directamente falso, *sin evaluar* su segundo argumento, y

v_c. que cuando el valor de su primer argumento es cierto ya devuelve directamente cierto, *sin evaluar* su segundo argumento.

- tipos numéricos **nat** (naturales), **ent** (enteros) y **real** (reales), con las operaciones aritméticas y relacionales habituales. Cuandos los números naturales o reales sean positivos, les asignaremos tipos nat^+ y $real^+$. Algunas veces conviene considerar dominios numéricos extendidos con valores infinitos; en tales ocasiones utilizamos los tipos nat_∞, ent_∞ y $real_\infty$
- tipo de caracteres **car**.
- tipos enumerados {$valor_1$,…,$valor_k$), con k≥1.
- rangos *i..j* donde *i* y *j* son números naturales. Si *i≤j*. el rango incluye todos los naturales desde *i* hasta *j*. ambos inclusive. Si *i > j*, el rango es vacío.
- *vectores:* Una variable V de tipo vector se declara V[í..j] **de** *tipo* con su rango además del tipo de sus elementos. A una posición *k* en el rango del vector, es decir, con *i≤k≤j*. se accede con la notación V[k], tanto para leer la información como para modificarla. Una asignación de la forma V[i..j] := [valor] modifica el contenido de todas las posiciones en el rango entre *i* y *j*. Los vectores pueden ser

de más de una dimensión.

- *registros*: Un tipo registro **reg** *$campo_1$:$tipo_1$,…,$campo_n$:$tipo_n$* **freg** se declara con sus campos y los tipos correspondientes. Si *R* es una variable de ese tipo, entonces la expresión *R.$campo_i$* accede a la información del campo con nombre *$campo_i$*, tanto para leerla como para modificarla.
- *punteros*, descritos en la Sección 2.1.2.

Por otra parte, como veremos a lo largo de este capítulo (y también en el resto de la primera parte del libro), en el lenguaje se definen los tipos de datos que se deseen a partir de los tipos básicos y de otros definidos anteriormente.

A la hora de escribir los algoritmos, distinguimos entre *funciones* y *procedimientos.* Una función se declara con varios parámetros de *entrada* (tal vez ninguno como caso extremo cuando la función es constante) y al menos un parámetro de *salida* (o resultado), cada uno de ellos con su tipo correspondiente.

fun **nombreFun**(e_1: *tipo1*,.... ,e_n: *$tipo_n$*) **dev** s: *tipo'*
var x_1: *tipo''$_1$*,…,x_k: *tipo''$_k$*
  P
ffun

Cuando la función tiene más de un parámetro de salida, estos se declaran de la forma ⟨s_1: *tipo'$_1$*,.... s_m:

tipo'$_m$⟩, con m>1.

Los parámetros de entrada no pueden cambiar a lo largo de la ejecución del cuerpo *P* de la función, por lo que. si el algoritmo modifica esos datos de alguna forma, es necesario realizar una copia (véase la Sección 2.1.3). En particular, no se permiten asignaciones a parámetros de entrada.

Además, en el cuerpo *P* se pueden usar variables auxiliares locales declaradas tras la cabecera. En general, no hacemos explícitas las declaraciones de variables auxiliares de tipos básicos, tales como *bool* o *nat* (como, por ejemplo, las variables usadas como contadores en bucles con contador), pero hay que tener en cuenta que todas las variables que no son parámetros de entrada o salida son variables auxiliares locales, y nunca hay variables globales.

Por otra parte, un procedimiento puede tener parámetros de *entrada/salida,* cuyo valor se puede modificar a lo largo de la ejecución, y parámetros de entrada que no cambian y que se declaran precediéndolos con el símbolo **e**.

proc nombreProc(**e** e_1: *tipo*$_1$,…,**e** e_n: *tipo*$_n$, es_1: *tipo'*$_1$,…,es_m: *tipo'*$_m$)
var x_1: *tipo''*$_1$,…, x_k: *tipo''*$_k$.
P
fproc

Aunque en este esquema aparezcan

primero los parámetros de entrada y luego los de entrada/salida, en general las diferentes clases de parámetros se pueden intercalar.

Cuando en la segunda fase de la implementación de un tipo de datos vamos a programar las operaciones correspondientes, en función de la representación concreta interna definida en la primera fase, hay que decidir para cada operación si se implementa como función o como procedimiento. En algunos casos, como por ejemplo para las operaciones observadoras. la elección está clara, dado que estas operaciones consultan una estructura de datos sin modificarla, y por tanto se componan claramente como funciones. Sin embargo, en otros casos la situación puede no ser tan clara. En general, nuestro enfoque es implementar como procedimientos aquellos algoritmos en los cuales una estructura se modifica e intuitivamente uno espera que el resultado sea la misma estructura modificada y no una copia modificada de la original. Por ejemplo, muchas estructuras (pilas, colas, listas, conjuntos, etc.) se comportan como “contenedores" con elementos que se añaden o se quitan, y en tales casos los "contenedores" no se copian. En cambio, cuando se hace la unión de dos conjuntos. la idea intuitiva es que

se tiene un nuevo "contenedor" con los elementos de las dos estructuras de partida y por tanto se implementa como una función.

En algunas ocasiones usamos la notación de *precondición* y *postcondición* para dar información sobre la entrada y la salida de un algoritmo. Específicamente, la precondición es una expresión booleana que expresa las condiciones sobre los parámetros de entrada de un algoritmo que garantizan que la aplicación del algoritmo tiene sentido, además de los tipos; por ejemplo, un algoritmo de división sobre un tipo numérico solo tiene sentido cuando el divisor no es nulo. La postcondición es una expresión booleana que relaciona los parámetros de salida con los de entrada, indicando de esta forma qué cálculo o proceso realiza el algoritmo sobre los datos de entrada.

2.1.2. Memoria dinamica

Como ya hemos mencionado al principio de esta introducción, vamos a construir implementaciones de estructuras de datos con memoria dinámica. La idea consiste en utilizar tipos definidos mediante registros que incluyan algún campo cuyo valor sea un puntero que señale a un registro del mismo tipo, y que será utilizado para "encadenar" posiciones de memoria.

Un ejemplo típico de tipos para representar *estructuras lineales enlazadas* es el siguiente:

```
tipos
enlace = puntero a nodo
nodo = reg
    valor : elemento
    sig : enlace
  freg
ftipos
```

Un valor de tipo *nodo* es un registro con 2 campos, en uno de los cuales se guarda cierta información de tipo *elemento* y en el otro hay un "enlace", que es un puntero a otro valor del mismo tipo *nodo.*

Nótese que esta definición de tipos es mutuamente recursiva puesto que el segundo campo de un nodo es un enlace que, a su vez, es un puntero a un nodo. El caso básico se obtiene cuando el puntero es nulo, es decir, tiene el valor constante nil que indica que el puntero no apunta a ninguna posición de memoria.

Como se **ve** en el ejemplo, para definir un tipo puntero se usa la notación **puntero a** *nombre-tipo.* Para acceder a los datos a los cuales apunta un valor *p* de tipo puntero, se usa la notación *p* f. que es entonces una expresión de tipo *nombre-tipo.*

Si *q* es una variable (o un campo de un registro o una componente de un vector) de tipo *enlace,* la instrucción reservaría) hace que el sistema operativo reserve, es decir, marque como utilizada, memoria suficiente para almacenar un valor de tipo *nodo* y hace que la variable puntero *q* apunte hacia la información en esa zona de memoria. De forma complementaria, la instrucción liberaría) hace que el sistema operativo marque como disponible la memoria del nodo apuntado por *q.*

2.1.3. Asignación, copiar y anular

Como se explicó en la Sección 2.1.1, en una instrucción de asignación x *:=* *E* el tipo de la variable x puede ser uno cualquiera del lenguaje, bien básico o bien definido por el usuario, siempre y cuando coincida con el de *E.* Para los valores de tipos básicos como *bool. nat.* etc., la ejecución de esta asignación supone que el valor de *E* se copia en la variable x. El coste en tiempo de realizar una asignación de esta clase es constante, considerando aparte el coste de evaluar la expresión *E* que en general

también será constante, salvo en el caso de llamadas a otros algoritmos y en particular de llamadas recursivas.

Para las construcciones básicas de tipos, corno vectores y registros, suponemos que una asignación entre dos expresiones del mismo tipo corresponde a un conjunto de asignaciones simultáneas para cada componente de la construcción. Por ejemplo, si V y W son vectores o registros del mismo tipo, la asignación *V :=* W es equivalente a una serie de asignaciones de cada componente de W en la componente correspondiente de V, al ir intuitivamente recorriendo las dos estructuras. Si se trata de un vector o registro con componentes cuyos tipos son básicos, una asignación de esta forma realiza la copia de todos los datos de W en *V* y su coste en tiempo es proporcional al tamaño de la estructura, a saber, el tamaño del rango del vector o el número de componentes del registro. En particular, el coste de una asignación de la forma V[i..j] := [valor] tiene coste proporcional al tamaño *j* - *i* + 1.

En cambio, para los tipos definidos por el usuario, una asignación x *:= E* sirve para denotar por x a la estructura *E.* Si esto supone o no una copia, no se sabe sin tener más información sobre la implementación concreta del tipo de *E.* lo cual al nivel abstracto se desconoce. Por ello, vamos a suponer que *no* se hace una copia y que puede haber

compartición de la estructura, es decir, que se permite que x "apunte" a la misma estructura en memoria que representa *E.* Cuando el usuario desea que en efecto tenga lugar una copia, por ejemplo para no modificar los datos de entrada de una función, se tiene que invocar explícitamente una operación copiar. Por lo que hemos comentado en los párrafos anteriores, tal operación no es necesaria para tipos básicos, vectores o registros, pero en la implementación de cualquier tipo abstracto de datos vamos a suponer que se tiene, además de las operaciones de la interfaz correspondiente, una operación para copiar una estructura de ese tipo.

En el caso de implementaciones estáticas basadas en vectores y registros, el algoritmo de copia puede consistir en una mera asignación, pues como acabamos de decir suponemos que el sistema de ejecución “sabe” cómo copiar tales construcciones de tipos. Sin embargo, en el caso de implementaciones dinámicas basadas en punteros, una mera asignación copiaría los punteros iniciales pero no el resto de la estructura, por lo que necesitamos programar el algoritmo de copia, que suele suponer un buen ejercicio de manejo de punteros y memoria dinámica. Por esta razón veremos vanos ejercicios de este estilo para diferentes estructuras de datos, empezando por el Ejercicio 2.8. En general, el coste en tiempo de copiar una

estructura dinámica es lineal con respecto a su tamaño, mientras que el coste de una asignación sin copia se puede suponer constante.

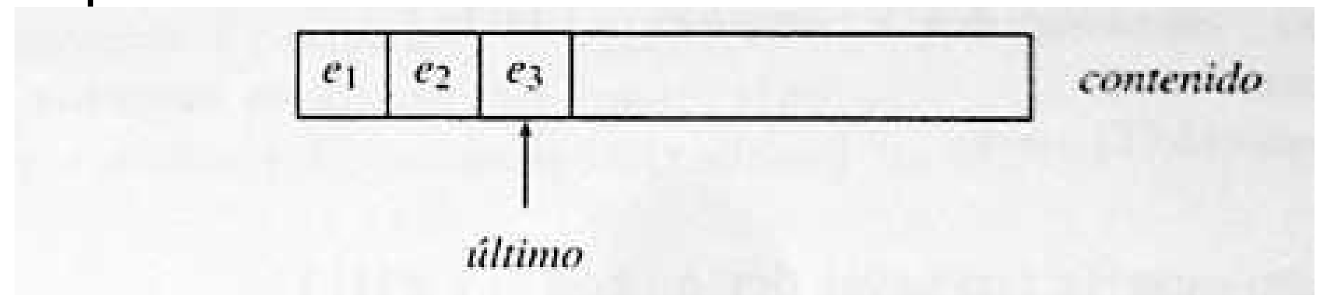


Figura 2.1: Representación de un conjunto mediante un vector.

Las estructuras de datos se van a utilizar en muchas ocasiones como estructuras auxiliares. Antes de dar por terminado cualquier algoritmo, es conveniente eliminar las estructuras auxiliares utilizadas, mediante la llamada a las correspondientes operaciones anular. Si bien la invocación de anular es independiente de la representación subyacente, su realización es especialmente importante cuando se trata de implementaciones dinámicas, pues así se libera toda la memoria ocupada por la estructura auxiliar. En muchos lenguajes esta gestión explícita de la memoria no es necesaria porque el sistema dispone de un mecanismo de "recogida de basura" *(garbage collection* en inglés), pero de nuevo consideramos que escribir estos algoritmos supone un buen ejercicio en el manejo de memoria dinámica que ayuda a entender la programación con punteros (véase el Ejercicio 2 8).

EJERCICIOS RESUELTOS

2.1. Implementar el TAD de los conjuntos finitos (Ejercicio 1.7) utilizando como representación del tipo de conjuntos

(a) vectores de elementos,

(b) vectores de elementos sin repeticiones, y

(c) vectores de elementos sin repeticiones y ordenados, suponiendo que el tipo de los elementos sobre el que se construyen los conjuntos admite una relación de orden total.

-----------------Solución----------------

La solución que sigue no está estructurada por apartados, sino que se organiza por algoritmos, destacando en cada uno las diferencias que aparecen entre las tres representaciones.

En las tres representaciones descritas en el enunciado los elementos se irán almacenando en un vector *contenido[* 1. JV], cuyo tamaño *N* está determinado *a príori* y en el cual no interesa dejar huecos entre las posiciones en las que se van almacenando los elementos, por lo que también necesitaremos tener un índice *último* que nos indique hasta qué posición tenemos ocupado el vector. Dicha representación se muestra gráficamente en la Figura 2.1.

Nótese que un conjunto es vacío si y solo si el índice *último* vale cero, y que se ignora por completo la información que pueda haber en el vector entre las posiciones *último* + 1 y *N.*

De esta forma, el tipo representante en los tres apartados es el siguiente registro con los dos campos que acabamos de describir:

tipos
conjunto = **reg**
contenido[l..tV| **de** *elemento*
último : 0../V
freg
ftipos

La implementación de las operaciones cjto-vacío. unit y es-cjto-vacío? es común y muy sencilla para las tres representaciones.

fun cjto-vacío() **dev** .v : *conjunto { (-)(!) }*
x.último := 0
ffun
fun unit(e : *elementó)* **dev** x : *conjunto* { 0(1) }
x.último **:= 1**
x.contenido[\] := e
ffun
fun es-cjto-vacío?(x : *conjunto)* **dev** *b : bool* (0(1))
b := (x.último **= 0)**
ffun

Para las tres representaciones la operación está? se puede implementar con un coste en *&(x .último)* (es decir, lineal con respecto al tamaño de la parte ocupada del vector), haciendo un recorrido del vector desde el principio hasta que o bien se encuentra el elemento buscado, o bien se llega al final de la parte ocupada. El algoritmo queda

como sigue:

fun está?(e : *elemento, x : conjunto)* **dev** *b : bool { ©(x.último))*

b := falso

i := **I**

mientras *i* < *x.último* A -•*b* **hacer**

b := (x.conreztíí7o[/'] = *e)*

i := *i* + 1

fmientras

ffun

Sin embargo, para la última representación, aprovechando que los elementos están ordenados, se puede obtener una implementación mucho más eficiente, con un coste en *0(\og(x .último)).* utilizando búsqueda binaria (véase el Ejercicio 11.1), para lo cual

(V[l] < ... < V[n] A l<c</ + l<n + l)

fun búsqueda-binaria(V[1 .JÍ] **de** *elemento, e : elemento, c. f : nat)* **dev** *(existe : bool. p : nat)*

{ (existe => c < p < f A V [p] = *e) /*

(-•existe => c < p < f 4- 1 A *V[c..p* — 1] < *e* < V[p../]) *}*

es una función de búsqueda binaria entre los índices *c y f.* donde *existe* es cierto si el elemento *e* se encuentra en V[c../] en la posición *p,* y *existe* es falso si el elemento *e* no está en V[c../], en cuyo caso *p* es la posición donde debería insertarse para mantener el vector ordenado (podría tener que insertarse al final, por eso *p* puede llegar a valer *f* 4- 1).

En los algoritmos que siguen, la letra al final del nombre se refiere al

apartado para el cual se aplican, según la representación.

fun está?-c(e : *elemento, x : conjunto)* **dev** *b : bool* **{ ©(log(x.úWmo)) }**

(b, n) := búsqueda-binaria(x.*contenido, e,* 1, *x.último)*

ffun

La implementación de la operación añadir depende de cada representación. En el primer caso basta añadir el elemento en la primera posición libre del vector (con un coste constante); en el segundo ocurre lo mismo, pero solo si el elemento no está ya en el vector, puesto que no puede haber repeticiones (el coste es entonces lineal por la búsqueda); y en el tercero, si no está el elemento, hay que insertarlo en la posición adecuada, para lo cual utilizamos la función búsqueda-binaria descrita anteriormente y un procedimiento auxiliar desplazar-der que desplaza elementos hacia la derecha del vector, siendo el coste total lineal con respecto a *x.último* (logarítmico por la búsqueda y lineal por el desplazamiento).

Nótese que debido al tamaño fijado a *príori* del vector subyacente *x .contenido))* ..TV], puede ocurrir que el conjunto esté lleno’ y que no se pueda añadir ningún elemento más; en tal caso, se devuelve un mensaje de error, que es propio de la implementación y no se debe

a la parcialidad de la operación, la cual es total al nivel abstracto de la especificación.

proc añadir-a(e *e : elemento, x : conjunto)* { 0(1))
si *x.último = N* **entonces** error(Espacio insuficiente)
sino *x.último := x.último* + 1 ; *x.contenido[x.último] := e*
fsi
fproc
proc añadir-b(e *e : elemento, x : conjunto) { ©(x.último) }*
si -■está?(e, x) entonces
si *x.último — N* **entonces** error(Espacio insuficiente)
si no *x.último := x.último* **+ 1 ;** *x .contenido[x .último] '.= e*
fsi
fsi
fproc

En el algoritmo para la tercera representación invocamos en primer lugar a la función búsqueda- binaría. Si el elemento *e* ya se encuentra en el vector, no hay que hacer nada: pero si el elemento *e* no se encuentra, hay que insertarlo en la posición *n* indicada por la búsqueda. Para realizar la inserción, supuesto que quede hueco libre para el nuevo elemento, se desplazan una posición hacia la derecha todos los elementos entre las posiciones *n y x.último.* invocando para ello al procedimiento desplazar-der.

proc añadir-c(e *e : elemento, x : conjunto) { ©(x.último) }*
(b. n) := búsqueda-binaria(.v.cozin?;nr/o. *e, 1, x.último)*

si -'*b* **entonces**
si *x.último = N* entonces error(Espacio insuficiente)
si no
desplazar-der(.v.*contenido, n. x.último)*
x,contenido[n] := e
x.último := x.último-}- 1
fsi
fsi
fproc
{l<c</+l</V + l}
proc desplazar-der(V[1 ../V] **de** *elemento,* e *c, f : nat) (&(f — c* + 1) }
para i' = *f* **+ 1 hasta** *c* + 1 **paso - 1 hacer**
V[/| := V[í - 1]
fpara
fproc

La implementación de la operación quitar depende también de cada representación. En el primer caso hay que recorrer el vector siempre hasta el final para asegurarse de quitar todas las posibles repeticiones, mientras que en el segundo podemos parar cuando se encuentra el elemento (o cuando se llega al final si no está). En ambos casos, para llenar el hueco que se deja al quitar un elemento, se coloca allí el último elemento.

proc quitar-a(e *e : elemento, x : conjunto) { ©(.x.último) }*
i := 1
mientras *i* < *x.último* **hacer**
si *x.contenido[i]* = *e* **entonces**

x .contenido[i] := x.contenido[x .último]

x.último := x.último — 1

si no *i :=* *i* + 1

fsi

fniientras

fproc

proc quitar-b(e *e : elemento,* ,r : *conjunto) { &(x.último)) i* := 1 **mientras** *i* < *x.último* Af *x.contenido[i] e* **hacer** *i :*= *i* + 1 **fmientras si** *i* < *x.último* **entonces** *x.comenido[i] :*= *x.contenido[x.último] x.último :*= *x.último* — 1 **fsi**
fproc

En el algoritmo para el tercer apartado, invocamos en primer lugar a la función búsqueda-binaria. Cuando el elemento *e* no se encuentra en el vector, no hay que hacer nada. Si el elemento *e* se encuentra en la posición *n* indicada por la búsqueda, se elimina llamando a un procedimiento auxiliar desplazar-izq que desplaza una posición hacia la izquierda todos los elementos entre las posiciones *n* + 1 *y x .último.* Aunque el coste de la búsqueda es logarítmico con respecto a *x.último.* el coste total es lineal debido al desplazamiento que hay que realizar.

proc quitar-c(e *e : elemento, x : conjunto) (* ©*(x.último)) {b, n*) := búsqueda-bínariaíx.conremr/r?, *e.* 1. *x.último)* **si** *b* **entonces**
desplazar-izqí.v.*contenido, n* + *\. x.último) x.último :*= *x.último* — 1
fsi fproc

(l<c</+l<ÍV + l)

proc desplazar-izq(V[1 ..TV] **de** *elemento,* **e** c, *f : nat)* (©(/ — c + 1) }
para *i* = *c* **hasta** *f* **hacer** V[í-1] := V[r]
fpara
fproc

Una interesante posibilidad para implementar las operaciones unión, intersección y diferencia es hacerlo en términos de cjto-vacío, está? y añadir, de manera que los algoritmos sean comunes para todas las representaciones, aunque el coste en tiempo dependerá del coste de las funciones y procedimientos invocados para cada representación.

fun uniónix, *y : conjunto)* **dev** *z : conjunto z* ■— cjto-vacio() **para** *i* = 1 **hasta** *x.último* **hacer** añadir(x.*contenido]!], z)* **fpara para** *i* = 1 **hasta** *y.último* **hacer** añadir(y.coriíenído[í]. z) **fpara**
ffun

fun intersección (,v. y : *conjunto)* **dev** z : *conjunto* Z := cjto-vacío() **para** *i* = 1 **hasta** *x.último* **hacer**
si *está?(x.contenido]!], y)* **entonces** añadir(x.conren/do[í], z) **fsi fpara**
ffun

fun diferencia(.r, *y : conjunto)* **dev** *z : conjunto*
z := cjto-vacíoQ
para *i* — 1 **hasta** *x.último* **hacer**
si -•*está?(x.contenido[i],* **y) entonces**
añadir(x.cwirevn</o[í],;) fsi
fpara
ffun

Vamos a analizar los costes en tiempo en el caso peor para cada representación.

(a) En la primera representación, los costes de crear el conjunto vacío y de añadir un elemento son constantes, mientras

que el coste del predicado está? es linea). Por tanto, el coste de unión está en ©*(x.último* 4- *y.último)* = 0(máx(x*.último, y.último))* al tratarse de dos bucles seguidos, y el coste de intersección y de diferencia están ambos en ©*(x.último y.último)* porque hay un único bucle con *x.último* iteraciones, en cuyo cuerpo se hace una búsqueda de coste en *Q(y .último)* para cada iteración.

(b) En la segunda representación, el coste de crear el conjunto vacío es constante, mientras que los costes de añadir y de está? son lineales. Como para hacer la unión se van añadiendo de uno en uno *x.último* 4- *y.último* elementos, el coste está en ©*((x .último* 4- *y .último)*^{2}*).* En los algoritmos intersección y diferencia, en el caso peor se añade un elemento en cada iteración del cuerpo del bucle, junto con una búsqueda de coste en ©*(y.último),* por lo que el coste de cada algoritmo está en ©*(x.último (y.último + x.último)).*

(c) En la tercera representación, el coste de crear el conjunto vacío es constante, el de añadir es lineal y el de está? es logarítmico. En el caso peor de la unión se añaden de uno en uno *x.último* 4- *y.último* elementos, por lo que el coste correspondiente está en ©*((x.último* 4- *y.último)-).* En los algoritmos intersección y diferencia, en el caso peor se añade un elemento en cada iteración del cuerpo del bucle, junto con

una búsqueda de coste en ® *()og(y.último)).* por lo que el coste de cada algoritmo está en esta ocasión en ©*(x.último* (log(y.*último)* 4- *x.último)).*

Finalmente, calcular el cardinal es inmediato en la segunda y tercera representaciones, porque todos los elementos son distintos y. por tanto, *x.último* cuenta exactamente el número de elementos en el conjunto.

fun cardinal-b/c(.v : *conjunto)* **dev** *n : nat* { 0(1))
n := x.último
ffun

En cambio, debido a las posibles repeticiones en la primera representación, el cálculo del cardinal es bastante más complicado. En el siguiente algoritmo cada elemento solamente se cuenta la última vez que aparece en el vector *contenido.* Para ello se busca entre las posiciones posteriores para ver si hay repeticiones; en caso afirmativo no se cuenta porque se hará luego, mientras que en caso negativo se cuenta el elemento. Otra posibilidad, que no detallamos por ser muy similar, es contar cada elemento la primera vez que aparece, viendo si hay repeticiones en posiciones anteriores. Debido a los dos bucles anidados, el coste es cuadrático con respecto a *x.último.*

fun cardinal-a1 (.v : *conjunto)* **dev** *n : nat { ©(x.último-) }*
n := 0
para *i* = l **hasta** *x.último* **hacer**

j := i 4- l
mientras *j < x.último* A(.
x.contenido[j\ x.contenido[i **hacer**
J := J 4- 1
fmientras
si *j > x.último* **entonces** *n ;= n* 4- 1
fsi
fpara
ffun

Una tercera posibilidad para la primera representación es utilizar un conjunto auxiliar *y* en el que solamente guardamos los elementos diferentes. Al final del procedimiento, como y es una versión optimizada de x en la que se han eliminado posibles repeticiones, el contenido de y se copia en x mediante una asignación entre registros, para mejorar así futuros accesos a x y reducir al mismo tiempo el espacio ocupado en el vector para su almacenamiento. Nótese que cardinal-a2 debe implementarse como procedimiento en vez de como función porque quita repeticiones al mismo tiempo que se calcula el cardinal. Sin embargo, el coste en tiempo permanece cuadrático en el caso peor.

proc cardinal-a2(x : *conjunto, n : nat) (*
&(x .último^{2}*))* **var y :** *conjunto*
y := cjto-vacío()
para i = 1 **hasta** *x.último* **hacer**
si -<*está?(.x.contenido[i].* y)
entonces añadir-a(.r.«w<?w<Zo[í], y)
fsi
fpara
n := y.último

x := y

fproc

Implementar el TAD de los conjuntos finitos (Ejercicio 1.7) sabiendo que los elementos son números naturales en el intervalo *1..N.*

Solución:

Al imponer requisitos muy exigentes sobre los elementos, podemos simplificar considerablemente la representación general de los conjuntos del Ejercicio 2.1, y utilizar ahora como representación de un conjunto x un vector x[l *..N]* de *booleanos.* de forma que Vi : 1 < *i* < *N :* A (Í | <=> *i* e A . En definitiva el vector representa explícitamente la *función característica* del conjunto representado.

tipos

conjunto **= vector** [1..A] **de** *bool*

elemento = \..N

ftipos

Para crear el conjunto vacío, todas las posiciones del vector se rellenan con el valor falso.

fun cjto-vacío() dev x : *conjunto { &(N))*

x[l..A] := [falso]

ffun

Para añadir un elemento, la posición del vector correspondiente a ese elemento se rellena con cierto (si el elemento ya estaba en el vector, la información no ha cambiado) y las demás se dejan igual. Para quitar un elemento se aplica la misma idea con falso.

proc añadir(e *e : elemento,* x : *conjunto)* { 0(1) }

x[e] := cierto
fproc
proc quitar(e *e : elemento, x : conjunto)*
(0(1) }
x[e] := falso
fproc

Para construir un vector unitario con el elemento *e,* podemos hacer x := cjto-vacío(); añadir(<?, x). Expandiendo ambos algoritmos, se obtiene el algoritmo que sigue.

fun unitfe : *elemento)* **dev** x *: conjunto {*
&(N))
x[l..7V] := [falso]
x[e] := cierto
ffun

En esta representación las búsquedas son inmediatas, pues basta acceder directamente a la posición correspondiente del vector.

fun está?(e : *elemento, x : conjunto)*
dev *b : bool* **{ ®(l) }**
b := x[e]
ffun

En cambio, para reconocer el conjunto vacío ahora es necesario recorrer todo el vector, puesto que la representación no contiene ningún "contador" que facilite información relacionada con la cardinalidad del conjunto representado. En el caso mejor, el coste es constante, pero en el caso peor el coste es lineal.

fun es-cjto-vacío?(x : *conjunto)* dev *b :*
bool (©(/V) |
i := l ; *b* := cierto
mientras *i* < *N* A *b* **hacer**

b := -x[t]
i - i + 1
fmientras
ffun

Las operaciones de unión, intersección y diferencia tienen una traducción inmediata en términos de operaciones booleanas.

- o Un elemento está en la unión cuando está en cualquiera de los dos argumentos, por lo que la unión corresponde a la disyunción.
- o Un elemento está en la intersección cuando está en ambos argumentos, por lo que la intersección corresponde a la conjunción.
- o Un elemento está en la diferencia cuando está en el primer argumento y no en el segundo, por lo que la diferencia corresponde a la conjunción con el segundo argumento negado.

fun unión(x, y : *conjunto)* **dev** *z : conjunto* (0(/V) }
para *i* — 1 **hasta** *N* **hacer**
z[í| := x[í] v y[í]
fpara
ffun
fun intersecciónix, y : *conjunto)* **dev** *z* ■ *conjunto { Q(N))*
para *i* = 1 **hasta** *N* **hacer**
<['] := -V[Í] A y[z]
fpara
ffun
fun diferenciatx, y : *conjunto)* **dev** *z* '■ *conjunto { &(N))*
para *i* = 1 **hasta** *N* **hacer**

```
    z[t] := x[í] A ->y[í]
  fpara
ffun
```

De nuevo, como la representación no contiene ningún "contador" explícito, para calcular el cardinal del conjunto representado hay que recorrer todo el vector, contando el número de posiciones cuyo valor es cierto.

fun cardinal(.v : *conjunto)* **dev** *n : nat { Q(N) } n* **:= 0**
para *i* **= 1 hasta** *N* **hacer**
si x[r] entonces *n := n* **+ 1 fsi**
fpara
ffun

2.3. Implementar el TAD de los multiconjuntos finitos (Ejercicio 1.8) suponiendo que los elementos son números naturales en el intervalo *1..N.*

------------------Solución------------------

Esta solución sigue las mismas ideas que las vistas en el Ejercicio 2.2 para la implementación de conjuntos finitos sobre el intervalo 1..7V. con la diferencia de que en vez de utilizar como representación un vector de booleanos. la representación de un multiconjunto x es un vector ,v[1. JV] de *números naturales* de forma que x[í] indica la *multiplicidad* en x del elemento *i.* para *i* entre 1 y *N,* puesto que no interesa saber solo si un elemento está o no, sino el número de veces que aparece.

tipos
multiconjunto = vector [1 ..Ai] de ***nat***
elemento = *\..N*
ftipos

Para crear el multiconjunto vacío, todas las posiciones del vector se rellenan con cero.

fun mcjto-vacío() dev x : *multiconjunto { Q(N)]*
x[I..A] := [0]
ffun

Para añadir un elemento, la posición del vector correspondiente a ese elemento se incrementa y las demás se dejan igual.

proc añadirle *e : elemento,* **x** *: multiconjunto)* **(0(1))**
x[e] := x[e] + 1
fproc

Para calcular la multiplicidad de un elemento dado, basta acceder directamente a la posición correspondiente del vector, pues la representación se basa precisamente en esa idea.

fun multip(e : *elemento,* **x :** *multiconjunto)* **dev** *n : nat* **{ 0(1))**
n **:= x[<?J**
ffun

En cambio, para reconocer el multiconjunto vacío hay que recorrer todo el vector, comprobando que todas las posiciones valen cero. En el caso peor, el coste es lineal, aunque en el caso mejor sea constante.

fun es-mcjto-vacío?(x : *multiconjunto)* **dev** *b : bool { &(N))*
i '■= **1 ;** *b* **:= cierto**
mientras *i < N A b* **hacer**
b := (x[í] = 0)
i := *i* **+ 1**
fmientras
ffun

Para quitar una aparición de un elemento en un multiconjunto. basta decrementar el valor de la posición correspondiente, pero solamente si el

elemento está en el multiconjunto. es decir, su multiplicidad es mayor que cero; cuando no está, el multiconjunto no se modifica. Para borrar (todas las apariciones de) un elemento en un multiconjunto. basta anular la posición correspondiente.

proc quitarte *c : elemento,* .v : *multiconjunto)* { 0(1))

si _r[e] > 0 **entonces** .r[e] := **.r[e| - 1 fsi**

fproc

proc borrarte *e : elemento, x : multiconjunto)* { 0(1))

x[e] := 0

fproc

Las operaciones de unión, intersección y diferencia tienen una traducción inmediata en operaciones aritméticas.

- Un elemento está en la unión de dos multiconjuntos tantas veces como en ambos, por lo que la unión corresponde a la suma.
- Un elemento está en la intersección de dos multiconjuntos el número de veces que está en el que menos, por lo que la intersección corresponde al mínimo.
- El número de veces que un elemento está en la diferencia de dos multiconjuntos se obtiene restando al número de veces que está en el primer argumento el número de veces que está en el segundo: cuando está más veces en el segundo, el elemento no aperece

en la diferencia.

fun uniónf.v, y : *multiconjunto)* **dev z :** *multiconjunto* { 0UV))

para *i* = 1 **hasta** /V **hacer**

=[(]:= -r[Z] + y[Z]

fpara

ffun

fun intersección (.v. y : *multiconjunto)* **dev** *z : multiconjunto* (0(/V) }

para *i* = 1 **hasta** *N* **hacer**

z[t] := min(.v[ZJ. y[Z])

fpara

ffun

fun diferenciad, y *multiconjunto)* **dev** *z : multiconjunto* (0(/V))

para *i* = 1 **hasta** /V **hacer**

si .r[Z| > y[ZJ **entonces** z[Z] := .v[Z] - y[í]

si no z[Z | := 0

fsi

fpara

ffun

El test explícito que aparece en el algoritmo anterior se puede eliminar si la implementación de la resta de naturales ya tiene en cuenta que al restar a un número otro mayor el resultado es cero.

Para calcular el "cardinal” de un multiconjunto. teniendo en cuenta las multiplicidades de los elementos. se recorre el vector sumando todas las multiplicidades que se van encontrando.

fun cardmal(.v : *multiconjunto)* **dev** *n : nat* { 0(/V))

```
  n := 0
  para i = 1 hasta N hacer
    n := n + A[Z1
  fpara
ffun
```

Para contar solamente los elementos distintos, también se recorre todo el vector, contando ahora el número de posiciones cuyo valor es un número mayor que cero.

```
fun cardinal-distfr : multiconjunto)
dev n : nat { @(N) }
   n := 0
   para i = 1 hasta N hacer
     si ,r[í] > 0 entonces n := n + 1
     fsi
   fpara
ffun
```

Los multiconjuntos sobre elementos 2.4. que admiten una relación de orden total se pueden implementar mediante tablas ordenadas, como veremos en el Ejercicio 7.9.

Implementar el TAD de los polinomios con coeficientes naturales (Ejercicio 1.9), representando un polinomio Jjf=₀c,x' 8ʳᵃd° *g* mediante un vector 7ᴶ[0../V] con *N* > g, de forma que /ᴶ[/ | = *ci* para 0 < *i* < *g.*

-------------------Solución----------------

De acuerdo con la idea del enunciado, el tipo representante consiste en un vector para guardar los coeficientes junto con información sobre el grado del polinomio, que coincide con el índice de la última posición “interesante".

```
tipos
  polinomio
  reg
    coefs\0..N] de nat
    grado : 0..N
ftipos
```

freg

Nótese que el rango del vector empieza en 0 en vez de en 1 como en otros ejercicios, y que los polinomios de grado 0 son los polinomios constantes, incluyendo el polinomio nulo. En particular, el coeficiente guardado en la posición del grado es distinto de 0. excepto para el polinomio nulo que tiene grado 0 y el número en esa posición es también 0.

La implementación de las operaciones poli-nulo y es-poli-nulo? es inmediata.

fun poli-nulo() **dev** *p : polinomio* (0(1))

p.grado := 0 : p.coe/s[0] := 0

ffun

fun es-poli-nulo?(p : *polinomio)* **dev**

b : bool (0(1) } i» := *(.P-grado* = 0)

A (p.coe/s[0] = 0)

ffun

Para sumar un monomio debemos distinguir casos para ver si el exponente correspondiente hace aumentar el grado o no. Esto puede causar un error si el tamaño del vector es demasiado pequeño.

En el caso mejor el coste es constante, pero en el caso peor el coste es lineal con respecto al valor del exponente que se añade, debido a la necesidad de recorrer parte del vector y rellenarlo con ceros.

proc sumar-mono(e

c, *i : nat, p :*

polinomio) { 0(í)) **si**

c **0 entonces**

casos
p.coefs[i] := p.coefs[i] + *c* (modificar coeficiente existente)
0 *p.grado* < *i* A *i* < *N* —>
p.coefs[p.grado + 1..Í — I] := [0]
{ nuevos coeficientes nulos }
p.coefs[i] := c { nuevo coeficiente no nulo)
p.grado := i { incrementar grado }
0 *i* > *N* —> error(Exponente demasiado grande)
fcasos
fsi
fproc

Como el polinomio resultado de sumar dos polinomios con coeficientes naturales tiene el grado igual al máximo de los grados de los dos argumentos, para sumar basta recorrer los dos vectores e ir sumando coeficiente a coeficiente; cuando se llega al grado mínimo, se copian los restantes coeficientes del polinomio de mayor grado. Por tanto, el coste de la suma es lineal con respecto al grado máximo. Nótese que implementamos sumar sin utilizar sumar-mono.

fun sumar(p, *q : polinomio)* **dev** *r : polinomio (p.grado. q.grado}))*
ni := mín(p.*grado, q.grado)*
r.grado : = máx(*p.grado, q.grado)*
para *i* — 0 **hasta** *ni* **hacer**
r.co<?/s[í] := *p.coefs[i]* 4- r/.coe/i[í]
fpara
{ copiar restantes de *p,* si los hay }

para *i = ni* 4- 1 **hasta** *p.grado* **hacer**
r.coefsfi] ;= p.coefs[i]
fpara
{ copiar restantes de *q,* si los hay)
para *i = ni* 4- 1 **hasta** *q.grado* **hacer**
r.coefs[i] := q.coefi[i]
fpara
ffun

En general, el producto de dos polinomios tiene el grado igual a la suma de los grados de los dos argumentos, pero cuando uno de estos es el polinomio nulo, el resultado es también el polinomio nulo; por eso. necesitamos tratar esa situación aparte. Para el caso general, vamos a ir sumando directamente (es decir, sin usar sumar-mono) en el vector resultado todos los monomios resultantes de todas las multiplicaciones de monomios entre los dos polinomios dados. De nuevo, hay que tener cuidado con que el polinomio resultante quepa en el vector.

fun multiplicarte, ¿/ : *polinomio)* **dev** *r : polinomio { <S)(p.gradoq.grado))*
si es-poli-nulo?(p) v es-poli-nulo?(r/) **entonces** *r* := poli-nulo()
si no
r.grado := p.grado 4- *q .grado*
si *r.grado > N* **entonces** error(Exponente demasiado grande)
si no
r.coefs[0..r.grado] := [0]
para *i* = 0 **hasta** *p.grado* **hacer**
para *j* = 0 **hasta** *q.grado* **hacer**

r.coefs[i 4- j] := *r.coefs[i* 4- j] 4- p.coe/s[í] * *q.coefs[j]*
fpara
fpara
fsi
fsi
ffun

Con esta representación los algoritmos para calcular un coeficiente y el grado de un polinomio dado son muy sencillos.

fun coeficiente^' : *nat. p : polinomio)* **dev** *c : nat* { 0(1))
si *i* < *p.grado* **entonces** *c* := p.coe/s[i]
si no *c* := 0
fsi
ffun

fun grado(p : *polinomio)* **dev** *g : nat* { 0(1) |
g p.grado
ffun

Finalmente, vamos a ver dos formas diferentes de implementar la evaluación de un polinomio sobre un número natural. En la primera utilizamos una variable auxiliar *pot* que va calculando las sucesivas potencias v^k del valor *v* dado, para *k* desde 0 basta el grado del polinomio.

fun evaluad (p : *polinomio, v : nat)* **dev** *w : nat { <r)(p.grado) }*
w := p.coe/s[0]
pot := 1
para *i* **= 1 hasta** *p.grado* **hacer**
*pot := pot * v*

w := u¹ + p.coe/s[í] * *pot*

fpara

ffun

Como se hace un recorrido del vector de izquierda a derecha, el coste es lineal con respecto al grado del polinomio, y el número de multiplicaciones que se hace es 2*p grado.*

En la segunda forma de evaluar, aplicaremos el *método de Hornee.* que se basa en la siguiente igualdad:

$C_g X^S$ + Cj-lX⁸"¹ + • ■ ■ + C2.V² + C|.V + C() =

(((. . . (.CgX + C_s-l).V + • • ■ + Cj).V + C2)Á **+ C|).V + <.'()**

fun evaluar2(p : *polinomio, v : nat)* **dev** *w* : nat **(0(p.***grado) }*

w := p.coefs] p.grado]

para *i* = *p.grado* — 1 **hasta** 0 **paso** — 1 **hacer**

w := *w* * *v* + p.coe/i[i]

fpara

ffun

Ahora se hace un recorrido del vector de derecha a izquierda, por lo que el coste continúa siendo lineal con respecto al grado del polinomio, pero el número de multiplicaciones se ha reducido a la mitad, siendo ahora igual a *p.grado.*

2.5. Modificar adecuadamente la implementación del TAD de los polinomios en la solución del Ejercicio 2.4 para que los coeficientes sean

números *enteros* en vez de naturales (tal como se especificó en el Ejercicio 1.10).

-------------------Solución----------------

El tipo representante consiste en un vector para guardar los coeficientes que ahora son enteros, junto con información sobre el grado del polinomio.

tipos

polinomio = **reg**

coe/s[0..Af] **de** *ent*

grado : 0../V

freg

ftipos

La diferencia importante con respecto a la implementación de polinomios con coeficientes naturales realizada en el Ejercicio 2.4 es que. en el caso de coeficientes enteros, no tenemos información *a priori* sobre el grado de una suma de polinomios, puesto que dos coeficientes con signo distinto se pueden anular al sumar y hacer disminuir el grado. Esto afecta a los algoritmos que implementan sumar-mono y sumar.

Vamos a reescribir los mismos algoritmos que en el Ejercicio 2.4. distinguiendo el caso adicional en el cual se pueden anular los coeficientes que determinan el grado. En este caso, recorremos el vector de derecha a izquierda, hasta encontrar el último coeficiente no nulo, o ver que el grado es cero. Los costes no cambian.

proc sumar-mono(e **c :** *ent,* **e** *i : nat, p : polinomio) ((-)(/))*

si c 0 entonces
casos
i < p.grado —» { modificar
coeficiente existente, grado no
cambia)
p.coefs[i] := p.coe/i[/] + c
0 *i = p.grado —>* { modificar
coeficiente existente y recalcular grado)
p.coefs\i] := p.coefs[i] + c
mientras *p.grado* **> 0** *A*
p.coefs[p.grado] = **0 hacer** *p.grado*
:= p.grado — 1 **fmientras**
0 *p.grado < i A i < N —>*
p.coefs[p.grado + l..í — 1] := [0|
{ nuevos coeficientes
nulos)
p.coe/s[z] := *c* (nuevo coeficiente
no nulo } *p.grado := i* {
incrementar grado }
0 *i > N* —> errorfExponente
demasiado grande) **fcasos**
fsi
fproc
fun sumar(*p. q polinomio)* **dev** *r :*
polinomio { tí(máx]p.grado. q.grado})
} ni := mín(*p.grado, q.grado)*
r.grado := máx(*p.grado, q.grado)*
para *i* = 0 **hasta** *ni* **hacer**
r.coefs]i] := p.coe/i[í] + ry.coe/sp']
fpara
casos
p.grado > ni —» { copiar restantes
de *p)*
r.coefs[m 4- *l.. p.grado] :=*
p.coefs[ni + 1 *.p.grado]*
0 *q.grado > ni —** { copiar restantes

de *q* |
r.coefs[ni + 1 *..q.grado]* *q.coefs[m*
4- La/.grado]
0 *p.grado* = *q .grado* —>{ recalcular grado)
mientras *r.grado* **> 0** A
r.coefs[r.grado] = **0 hacer** *r.grado*
:= r.grado — **1**
fmientras
fcasos
ffuii

En la operación para multiplicar no hay que hacer ningún cambio, puesto que el grado del polinomio producto de dos polinomios no nulos continúa siendo la suma de los grados de los dos argumentos, ya que al multiplicar dos números distintos de cero nos aseguramos que el coeficiente que determina el grado no se puede anular (el caso de multiplicar por un polinomio nulo se trataba por separado y tampoco cambia).

La implementación de la operación poli-nulo es obviamente la misma, y las demás operaciones son observadoras que funcionan de la misma forma independientemente del tipo de los coeficientes (como evaluar, por ejemplo).

2.6. Implementar el TAD de los polinomios con coeficientes naturales (Ejercicio 1.9). representando cada polinomio mediante un vector de pares, de manera que cada par esté formado

por un coeficiente y un exponente, y el vector esté ordenado de menor a mayor según el exponente.

------------------Solución----------------

A la hora de implementar los polinomios, la alternativa más simple para representar un polinomio ^*2f*=*o*$^{c'x'}$ grado *g* es utilizar un vector P[0..TV] con *N* > *g.* de forma que *P*[í] = c, para 0 < *i* < g, como se ha hecho en el Ejercicio 2.4. Sin embargo, esta representación es poco apropiada cuando muchos coeficientes c, son nulos. Consideremos, por ejemplo, el polinomio x^{1000}; con la representación anterior, necesitamos 1000 posiciones con el valor 0 y una posición con el valor 1. Si el tamaño *N* del vector que se fija *a priori* es demasiado pequeño, ni siquiera lo podemos representar. Por el contrario, la representación de este polinomio requiere una única posición si la representación solo guarda los monomios no nulos. Naturalmente, en este caso desaparece la relación entre exponente y posición del vector, por lo que junto a cada coeficiente tendremos que guardar el exponente al que va asociado. De esta forma, para representar un polinomio podemos utilizar un vector de pares *{coeficiente, exponeitte),* donde solo guardamos los monomios no nulos y no hay dos monomios con el mismo exponente. Para localizar más rápidamente la posición que ocupa un

monomio, imponemos la propiedad adicional de que el vector esté ordenado según los exponentes. Así, el tipo representante se define como sigue:

tipos
monomio = **reg**
coeficiente : naf[1-]
exponente : nat
freg
polinomio = **reg**
monomios[l..N] **de** *monomio* { ordenado según el campo *coeficiente)*
último : 0..N
freg ftipos

La implementación de las operaciones poli-nulo y es-poli-nulo? es inmediata.

fun poli-nulo() **dev** *p : polinomio* (0(1) }
p. último := 0 **ffun**
fun es-poli-nulo?(p : *polinomio)* **dev** *b : bool* { 0(1))
b := (p.último = 0)
ffun

Para la implementación de las operaciones sumar-mono y coeficiente utilizamos una función auxiliar de búsqueda binaria (véase el Ejercicio 11.1) buscar-exp. que busca (con coste en tiempo logarítmico con respecto al tamaño del vector) un exponente dado en un vector que representa un polinomio, cuyos elementos están ordenados según el campo que denota el exponente. Aunque en esta ocasión los datos en el vector son registros con varios campos, y

la ordenación y búsqueda se hacen con respecto a uno de los campos (en este caso, el campo *exponente),* el algoritmo tiene exactamente la misma estructura que búsqueda-binaria, si bien difiere en algunos detalles que nos impiden invocar directamente a este último. Por ello incluimos a continuación el código completo de la operación.

(P[1..(V] ordenado según *exponente* A l<c</+l</V+l)

fun buscar-exp(P[1 *..N* **] de** *monomio,*
i. c. f -.nat) **dev (***existe : bool. p : nat)*
casos
c > f —> (existe, p) := (falso, c)
□ *c < f —> ni := (f* — c + I) div 2
casos
i< *P[m].exponente -> (existe, p)*
:= buscar-exptP. *i. c. m* — 1)
□*i=* *P [ni (.exponen te —**
(existe, p):=(cierto, *ni}*
0*i>* *P [ni [.exponen te —>*
(existe, p}:=buscar-exp(*P. i. ni* -I- 1.
f)
fcasos
fcasos
ffun

((existe => c <p < f P\p].exponente = i) A
*{-•existe => c < p < f + * A *P[c..p — 1 [.exponente < i < P\p..f [.exponente)* |

Para implcmentar sumar-mono usamos buscar-exp para localizar la posición donde debe almacenarse el nuevo monomio, y un procedimiento auxiliar desplazar-der (como en la

implementación de los conjuntos en el Ejercicio 2.1) para efectuar desplazamientos en el vector de monomios, en el caso en que sea necesario insertar un par con un nuevo exponente (en particular, nótese que cuando el monomio es nulo no hay que hacer nada). Si el vector estuviera lleno, la inserción no podría hacerse produciéndose un error. El coste en tiempo de la búsqueda es logarítmico, pero el del desplazamiento es lineal, por lo que el coste total es lineal con respecto a *p.último.*

proc sumar-mono(e *c. i : nat. p : polinomio) { ©(p.último))*
  si *c* 0 **entonces**
    (b.n) := buscar-exp(p.*monomios, i*, 1. *p.último)*
    si *b* entonces { modificar coeficiente no nulo existente)
      p.monomios[n].coeficiente :=
      p.monomios[n].coefieiente + *c*
    si no { añadir nuevo exponente |
      si *p.último* = *N* **entonces**
      error(Espacio insuficiente)
      si no
        desplazar-der(*p.monomios, n. p.último)*
        p.monomios[n[.coeficiente := c
        p.monomios[n[.exponente := i
        p.último := p.último + 1
      fsi
    fsi
  fsi

fproc

La operación modificadora sumar se implemento en términos de sumar-mono. El primer bucle copia el polinomio *p* en *r* y luego el segundo va sumando los monomios de *q* al polinomio *r.*

fun sumar(p, *q : polinomio)* **dev** *r : polinomio*

r.último := p.último

r.mononiios] \..r.último] := p.monomios] l..p. último]

para *j* = 1 **hasta** *q.último* **hacer**

sumar-mono(</.monomí<).s[_/].coe/ici<'nre, *q.monomios] j[.exponente, r)*

fpara

ffun

El coste de la copia es lineal con respecto a *p.último.* A continuación, en el caso peor, todos los elementos que se añaden mediante sumar-mono se insertan a la izquierda de los elementos de *p* que necesitan ser desplazados, lo que supone un coste en tiempo en *(©(p.último q.último).* Sin embargo, si el tamaño de *p* fuera mucho menor que el de *q,* el coste de la búsqueda podría ser peor que el coste del desplazamiento. Como la suma es conmutativa y por tanto no importa el orden de los argumentos, **vamos a** suponer que el tamaño de *p* es mayor que el de *q* para concluir que el coste total está en **O(p,** *último q. último).*

Para la operación modificadora

multiplicar, podríamos utilizar una operación auxiliar mult-mono, como **hicimos** en la especificación (véase la solución del Ejercicio 1.9). pero en lugar de eso vamos a sumar al resultado todos los monomios resultantes de realizar las multiplicaciones entre monomios de los dos polinomios dados, pero ahora invocando a la operación sumar-mono, a diferencia de la implementación realizada en el Ejercicio 2.4.

fun multiplicaría, *q : polinomio)* **dev** *r : polinomio*

r := poli-nulo()

para *i* **= 1 hasta** *p.último* **hacer**

para *j* = 1 **hasta** *q.último* **hacer**

*sumar-mono(p.monomios[i].coeficiente * q .monomios] j].coeficiente, p.monomios[i].exponente* -I- *q.monomios[j].exponente.* r)

fpara

fpara

ffun

El coste está en ©*((p.último q .último)-),* debido a los dos bucles anidados y en base al tamaño del vector en el que se suma.

Para calculare! coeficiente asociado a un exponente basta hacer una búsqueda del exponente; si no se encuentra, entonces el coeficiente es nulo. El coste es logarítmico por la búsqueda.

fun coeficiente^' : *nat. p : polinomio)* **dev** *c : nat* *{ ©(\og(p.último)) |*
(b.n) := buscar-exp(p.monomios, i, 1. *p.último)*
si *b* **entonces** *c :=* *p.monomios[n].coeficiente*
si no *c* := 0
fsi ffun

Para calcular el grado de un polinomio no nulo basta con mirar el exponente mayor en el vector que representa el polinomio, que se encuentra en la posición indicada por *último.*

fun gradofp : *polinomio)* **dev** *g : nat* { 0(1))
si *p.último* / 0 **entonces** g ;= *p.monomios[p.último].exponente*
si no *g* **:= 0**
fsi ffun

En la siguiente función para evaluar un polinomio sobre un número natura!, utilizamos la función exp que implementa la operación de exponenciación sobre naturales. El coste del algoritmo depende del coste del algoritmo de exponenciación (que podría ser lineal o logarítmico con respecto al valor del exponente).

fun evaluar1(p ; *polinomio, u : nat)* **dev** *w : nat*
w := 0
para *i* = **1 hasta** *p.último* **hacer**
*w := w + p.monomios[i].coeficiente ** exp(u, *p.monomios[i].exponente)*
fpara

ffun

Una implementación alternativa, que no invoca ningún algoritmo de exponenciación. consiste en utilizar variables auxiliares *j y pot* para llevar calculada en *pot* la potencia r' del valor dado. El coste **en** este caso es lineal con respecto al grado del polinomio.

fun evaluar2(p : *polinomio, v : nat)* **dev** tu : *nat* { ©(grado(p)l }
w := 0 ; *j* := 0 ; *pot :*= I
para *i* = **l hasta** *p.último* **hacer**
mientras *j p.monomios[i].exponente* **hacer** { exponentes ausentes)
j := j + 1 ; *por := pot * v*
fmientras
*w w + p.monomios\i].coeficiente *pot*
fpara
ffun

Aunque este segundo algoritmo lleva precalculadas las potencias de r para no tener que recalcularlas partiendo de cero, no siempre es más eíiciente, pues puede ocurrir que el polinomio sea poco "denso", en el sentido de que tenga pocos coeficientes no nulos y cuyos exponentes sean elevados. En tales casos extremos es mejor hacer unas pocas llamadas a un algoritmo logarítmico de exponenciación (véase el Ejercicio 11.19) que calcular todas las potencias.

Un;t tercera posibilidad, que nos limitamos a mencionar, es utilizar el *método de Horner* para imple- mentar el

procedimiento de evaluación (véase la solución del Ejercicio 2.4).

2.7. Modificar adecuadamente la implementación del TAD de los polinomios en la solución del Ejercicio 2.6 para que los coeficientes sean números *enteros* en vez de naturales (tal como se especificó en el Ejercicio 1.10).

------------------Solución----------------

Aparte de cambiar los tipos adecuadamente, la única operación cuya implementación hay que cambiar es sumar-mono, que ahora utiliza otro procedimiento auxiliar desplazar-izq (véase la solución del Ejercicio 2.1) para efectuar desplazamientos a la izquierda en el vector de monomios cuando un coeficiente se anula, pues en el vector solamente se guardan los monomios no nulos.

proc sumar-mono(e *c : ent.* e *i : nat. p : polinomio) { p.último }*
si c / O **entonces**
(b. n) := t>usca-exp(p.monomios, i. 1. *p.último)*
si *b* **entonces**
p.monomios[n [coeficiente := p.monomios[n].coeficiente + c
si *p.monomios[n],coeficiente = G* **entonces**
desplazar-izq(p.mono/nío.s. *n* + 1. *p.último)*
p.último := p.último — 1
fsi
si no

si *p.último* = *N* **entonces**
error(Espacio insuficiente)
si no
desplazar-der(p.*monomios, n. p.último)*
p.monomios[n].coeficiente := c ;
p.monomios[n].exponente := i
p.último := p.último + 1
fsi
fsi
fsi
fproc

Figura 2.2: Estructura enlazada lineal.

Las demás operaciones, o bien están implementadas en términos de sumar-mono, como sumar y multiplicar, con lo cual ya se gestiona indirectamente la anulación de coeficientes, o bien son operaciones que no dependen del tipo de los coeficientes y, por tanto, funcionan exactamente de la misma forma.

2.8. Consideremos una estructura dinámica enlazada lineal como la de la Figura 2.2. Escribir algoritmos genéricos para *copiar* y *anular* una estructura enlazada lineal de dicha forma, sin suponer conocido el tipo *elemento* de los datos que se guardan en cada nodo.

------------------Solución----------------

El tipo que representa la estructura enlazada lineal se define de la siguiente manera:

tipos
enlace **= puntero a** *nodo*
nodo **= reg**
valor : elemento
sig : enlace
freg
estructura = enlace
ftipos

Como se ha comentado en la Sección 2.1.2. esta definición de tipos es mutuamente recursiva.

La idea del algoritmo para copiar es que se construya en *q* una copia idéntica de la estructura *p* dada como argumento. Obsérvese que una asignación *q := p* no tendría este efecto, ya que solo se haría una copia de punteros, es decir, los punteros *p y q* apuntarían a la misma estructura, pero no habría una duplicación de la memoria ocupada, de forma que *q* apunte a una estructura idéntica a la que es apuntada por *p* pero en posiciones de memoria diferentes. Primero vemos una versión recursiva de copiar.

fun copiar*(p : estructura)* **dev** *q : estructura* **var** *r : estructura*
si *p* = nil **entonces** *q* := nil
si no
r := copiar(/r f *.sig)* { copiar recursivamente el resto de la estructura)
reservar(q)
q f *.valor :=* copíar-elem(p f *.valor)* (copiar elementos)
q f *.sig := r*

fsi

ffun

Destaquemos la invocación que se hace de la función copiar-elem para copiar los valores guardados en los nodos. Dependiendo del tipo de tales valores, puede que la copia se haga con una simple asignación, con una llamada recursiva al mismo algoritmo copiar (cuando el tipo *elemento* coincide con el tipo *estructura,* en el caso de estructuras anidadas de la misma clase) o con un algoritmo de copia semejante para otra clase de estructuras (en el caso de estructuras anidadas de diferentes clases).

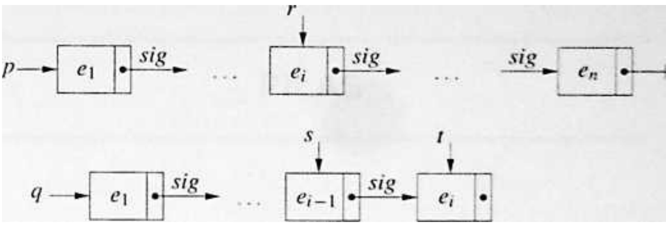


Figura 2.3: Copia iterativa de una estructura enlazada.

Una versión iterativa del algoritmo copiar es como sigue. El puntero *r* recorre la estructura *p.* En el caso no nulo, después de crear el primer nodo al que se apunta con *q.* se van creando los nodos siguientes, apuntados por *t.* y se van enlazando con el anterior, al que se apunta con s. Cuando se llega al final de />, es decir, cuando *r* = nil, se termina también la estructura copiada con otro puntero nulo.

fun copiar-ít(/; : *estructura)* **dev** *q : estructura*

var *r,s,t : enlace*
si *p* = nil **entonces** *q* := nil
si no
r := p
reservarte/)
q J *.valor* := copiar-elem(r f *.valor)* (copiar elementos |
5 *:= q*
mientras *r* f *.sig* nil **hacer**
r := r'f.sig
reservar(z)
i'{.valor := copiar-elem(r f *.valor)* (copiar elementos }
5 t *.sig := t*
s := t
fniientras
5 f *.sig :=* nil
fsi ffun

El procedimiento anular recorre las posiciones de la estructura liberando uno a uno cada nodo. Por si se diera el caso de que los elementos que se guardan en cada nodo necesitaran ser anulados de forma semejante, se invoca al procedimiento anular-elem que libera la memoria correspondiente. Este procedimiento puede ser el mismo anular, otro semejante para otra clase de estructuras, o simplemente vacío (es decir, un procedimiento que no hace nada) en el caso de datos de tipos básicos como *nat* y *bool.* o construcciones básicas de tipos como vectores y registros.

proc anular(p : *estructura)*

var *q : enlace*
mientras *p* nil **hacer**
q ■= *P* • *P* ■= *Pt -sis*
anular-elem(<y f *.valor)* { anular elementos |
liberaría)
fniientras
fproc

Capitulo 3

3. PILAS

En la vida cotidiana suele ocurrir que las tareas se nos acumulan de forma que siempre nos dedicamos a la tarea más reciente, dejando aplazadas las más antiguas, es decir, siguiendo el criterio de *el último en entrar es el primero en salir*.

En este capítulo vamos a considerar una estructura de datos lineal cuya característica principal es que el acceso a los elementos se realiza en orden inverso al de su almacenamiento. Aunque se las puede denominar estructuras ***LIFO*** *(Last In, First Oía* en inglés), el nombre más usual, que es el que vamos a utilizar, es ***PILAS***. El comportamiento de las pilas es totalmente independiente del tipo de los datos almacenados en ellas, por lo que se trata de un tipo de datos parametrizado.

La ventaja de las pilas es que el acceso a la estructura, tanto para su modificación (inserción y borrado) como para la consulta de los datos almacenados, se realiza en un único punto (la *cima* de la pila), lo que facilita implementaciones sencillas y eficientes.

A pesar de su sencillez, es una estructura con múltiples aplicaciones en el diseño de algoritmos. como la evaluación de expresiones o la implementación de la recursión.

EJERCICIOS RESUELTOS

3.1. Especificar un TAD para describir las *pilas* con elementos pertenecientes a un tipo dado como parámetro, con las siguientes operaciones:

- crear la pila vacía,
- apilar un elemento,
- desapilar el elemento en la cima.
- consultar el elemento en la cima, y
- determinar si la pila es vacía.

--------------------Solución:--------------------------

Como ya se ha comentado en la introducción, el comportamiento de una pila es independiente del tipo de los datos almacenados en ella. Ello significa que no se requiere ninguna característica especial para el tipo parámetro, por lo que consideraremos el parámetro más simple. *ELEM.* definido en la Sección 1.1.5.

La elección de constructoras no ofrece muchas alternativas: para obtener una pila se parte de una estructura vacía (pila-vacía) y se van apilando los elementos (apilar). Como el orden de apilación es fundamental para la posterior consulta y eliminación, las constructoras son *libres.*

El comportamiento de las otras tres operaciones (una modificadora y dos observadoras) viene dado mediante la distinción de casos de las dos constructoras. Las operaciones de consulta y eliminación del elemento en la cima de una pila solo tienen sentido si la pila no es vacía, por lo que ambas

operaciones son *parciales.* Las ecuaciones correspondientes simplemente reflejan el comportamiento LIFO de una pila: la cima corresponde al último elemento apilado. Entonces, en el caso no vacío, cima y desapilar son las “destructoras" asociadas a la constructora apilar.

especificación *PILASfELEM]*
usa *BOOLEANOS* **tipos** *pila*
operaciones pila-vacía apilar
desapilar cima es-pila-vacía?
variables
e : elemento p : pila *elemento pila*
ecuaciones *pila pila pila*
desapilar(pila-vacía)
desapilar(apilar(<?, p))

|> *pila* (constructora) (
pila constructora)
pila
elemento
bool

error
P

cima(pila-vacía) = error
cima(apilar(e. /;)) = *e*
es-pila-vacía?(pila-vacía) = cierto
es-pila-vacía?(apilar(<?. /;)) = falso
fespecificación

Extender la especificación de las pilas del Ejercicio 3.1 con las siguientes operaciones:

- calcular el número de elementos en una pila (profundidad).
- consultar el elemento del fondo de una pila,
- obtener la inversa de una pila,
- duplicar una pila p, de forma que cada elemento de p aparezca apilado dos veces seguidas, conservando el mismo orden relativo que en p,
- concatenar dos pilas, colocando los elementos de la segunda sobre los de la primera, y
- entremezclar dos pilas, donde la pila resultante se obtiene apilando alternativamente los elementos de las pilas argumento, empezando por el fondo de la primera pila.

------------------Solución-------------------

De forma similar a la operación cima (véase el Ejercicio 3.1). la consulta del elemento en el fondo de una pila solamente tiene sentido si la pila no es vacía, por lo que esta operación es parcial.

La especificación del comportamiento de todas estas nuevas operaciones se hace, en general, por distinción de casos sobre las dos constructoras de las pilas (pila-vacía y apilar), pero para facilitar la especificación de inversa, utilizaremos una operación auxiliar apilar-inversa con dos argumentos de tipo *pila,* que coloca la inversa de la primera sobre la segunda.

especificación *PILAS+[ELEM]*
usa *PILAS[ELEM], NATURALES*
operaciones

[illegible]

variables
e. f : elemento p.q : pila

La profundidad de una pila vacía es cero; cada vez que se apila un elemento, la profundidad de la pila se incrementa en uno.

ecuaciones
profundidad(pila-vacía) = 0
profundidad(apilar(e, p)) = I + profundidad(p)

El acceso a la cima de la pila es directo, de ahí la sencillez de las ecuaciones dadas en la especificación de las pilas para cima y desapilar (Ejercicio 3.1); pero para poder acceder al fondo de la pila hay que “avanzar" a través de los elementos apilados hasta alcanzar el primero que se apiló. Por eso se distinguen dos casos cuando la pila no es vacía: si hay solo un elemento o hay más de uno.

fondo(pila-vacia) = error
fondo(apilar(e. p)) = *e* <= es-pila-vacía?(p)
fondo(apilar(e. p)) = fondo(p) <= -’es-pila-vacía?(p)

Para apilar la inversa de una pila sobre otra, basta con ir desapilando los elementos en la cima de la primera y

apilarlos sobre la segunda.

apilar-inversa(pila-vacía, p) = p

apilar-inversa(apilar(e, p). r/) = apilar-inversa(p. apilarte. <?))

Entonces, para invertir una pila, nos limitamos a apilar su inversa sobre una pila vacía.

mversa(p) = apilar-inversa(p. pila-vacía)

Para duplicar los contenidos de una pila, basta apilar dos veces la cima de la pila sobre el resto de la pila, el cual ha sido previamente duplicado.

duplicar(pila-vacía) = pila-vacía

duplicar(apilar(e, p)) = apilarte, apilarte. duplicar(p)))

Para especificar la concatenación, distinguimos casos sobre la *segunda* pila a concatenar: si es vacía, nos quedamos con la primera pila, pero si no lo es. tendremos que apilar la cima de la segunda pila sobre la concatenación de toda la primera pila con el resto de la segunda.

concatenar(p, pila-vacía) = p

concatenarfp. apilarte, r/)) = apilarle. concatenar(p. </))

Una opción alternativa es colocar sobre la primera pila la inversa de la segunda pila invertida, utilizando las operaciones apilar-inversa e inversa (nótese el cambio en el orden de los argumentos).

índice-cinta

Figura 3.1: Representación estática de una pila.

concatenaría, *q)* = apilar-inversa(inversa(r/)./>)

Para entremezclar dos pilas, hay que comparar su profundidad, pues el último elemento a apilar será el de la cima de la pila con mayor profundidad y, en el caso de que ambas sean igual de profundas, corresponderá a la cima de la segunda pila.

entremezclar(pila-vacía. pila-vacía) =		pila-vacía
entremezclar(apilar(e, /?), *q)*	=	apilar(e. entremezclaría. *q))*
		profundidad(p) + 1 > profundidadír/)
entremezclaría. apilar(e. </))	—	apilarte, entremezclare/?. </))
	<=	profundidad!p) < profundidad(</) + I

fespecificación

aaaaaaaaaa

3.3. Diseñar una representación estática del TAD de las pilas utilizando vectores e implementar las operaciones especificadas en el Ejercicio 3.1.

--------------------Solución----------------

La idea es utilizar un vector *contenido* donde vamos colocando los elementos de la pila, de izquierda a derecha según se van apilando, y un índice que apuntará a la cima, como muestra gráficamente la Figura 3.1. en la cual se representa la pila de naturales

apilar(3. apilar(0. apilar(8. apilar(5. pila-vacía))))

Por tanto, el tipo representante es el siguiente:

tipos
pila = reg
contenido{ 1 *..N]* de *elemento*
índice-cinta : 0..N
freg
ftipos

Cuando la pila esté vacía, *índice-cinta* tendrá el valor cero.

Como ya se comentó en la Sección 2.1. estas representaciones sobre memoria estática tienen el inconveniente de que el tamaño de la estructura queda restringido por la capacidad del soporte de almacenamiento, en este caso, por el tamaño del vector *contenido.* De esta forma, la operación de apilar resulta ser parcial, produciéndose un error cuando se intenta rebasar dicha capacidad.

La implementación de las operaciones es la siguiente:

fun pila-vacía() **dev** *p : pila* (0(1) }
  p.índice-cinta := 0
ffun

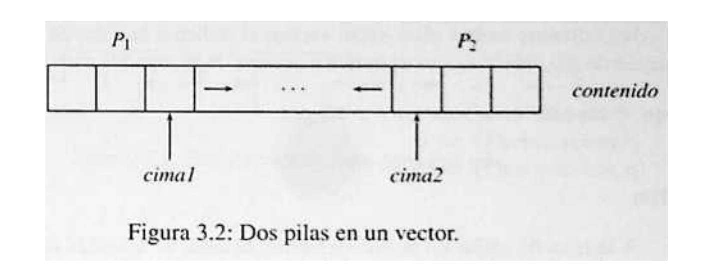


Figura 3.2: Dos pilas en un vector.

proc apilar(e *e : elemento,* p : *pila)* { 0(1) }
si *p.índice-cima = N* **entonces** error(Espacio insuficiente)
si no
p.índice-cima := p.índice-cima + 1
p.contenido[p.índice-cima] := e
fsi
fproc
proc desapilar(p : *pila)* **(0(1) }**
si *p.índice-cima* = 0 **entonces** error(Pila vacía)
si no *p.índice-cima := p.índice-cima* — 1
fsi
fproc
fun cima(p : *pila)* **dev** *e : elemento* { 0(1))
si *p.índice-cima* **= 0 entonces** error(Pila vacía)
si no *e := p.contenido] p.índice-cima]*
fsi
ffun
fun es-pila-vacía'fp : *pila)* { 0(1) } **dev** *b : bool*
b := (p.índice-cima = 0)
ffun

3.4. Implementar dos pilas sobre un mismo vector, de forma que el desbordamiento de cualquiera de las

pilas solo se produzca en el caso de que el vector esté completamente ocupado.

Solución:

A partir de la representación estática mediante un vector y un índice (véase el Ejercicio 3.3), la idea es utilizar el mismo vector de almacenamiento para las dos pilas, de forma que una va avanzando desde el extremo izquierdo, mientras que la otra lo hace desde el extremo derecho, como ilustra el gráfico de la Figura 3.2. De esta forma, el soporte estará completamente lleno cuando los dos índices tengan valores consecutivos.

Así pues, el tipo representante es el siguiente:

tipos
doble-pila = **reg**
contenido] 1 ../V] **de** *elemento*
índice-cima] 1..2] **de** 0..ÍV + 1
freg
ftipos

Inicialmente ambas pilas están vacías; el índice a la cima de la primera estará más allá del extremo izquierdo (0), mientras que el índice a la cima de la segunda estará más allá del extremo derecho *(N* + 1).

fun doble-pila-vacía() **dev** *p : doble-pila*
p.índice-cima[l] := 0
p.índice-cima[2] := *N* + 1
ffun

A la hora de apilar un nuevo elemento. la cima se desplaza una posición: hacia la derecha en el caso de la primera pila (+1) y hacia la izquierda en el caso de la segunda pila (— 1). En cambio, para desapilar, el desplazamiento de la cima se realiza en el sentido contrario: hacia la izquierda en el caso de la primera pila (—1) y hacia la derecha en el caso de la segunda pila (+1). Evitamos el uso de una instrucción condicional mediante la exponenciación del valor — 1: $(—l)^1$ = -1 y (- 1)- = +1.

En los cuatro algoritmos que siguen, el argumento *i* de tipo I..2 indica sobre cuál de las dos pilas tiene lugar la acción correspondiente.

proc doble-apilar(e *e : elemento, p : doble-pila,* e *i* : 1 ..2) { 0(1) }
si *p.índice-ciina[\] + 1 = p.índice-ciina[2]* **entonces** error(Espacio insuficiente)
si no
p.tndice-cima[i] := p.índice-cima[i] — exp(—1, z)
p.contenido[p.índice-ciina[i]] := e
fsi
fproc

proc doble-desapilar(*p : doble-pila,* **e** *i :* 1..2) (0(1) }
si doble-es-pila-vacía?(p. *i)* **entonces** error(Pila vacía)
sino *p.índice-cima[i] := p.índice-cima[i]* + exp(— 1.z)
fsi

fproc

fun doble-cima(/? : *doble-pila, i* : 1..2) **dev** *e : elemento* (0(1))

si doble-es-pila-vacía?(p. z) **entonces** error(Pila vacía)

sino *e := p.co>uenido[p.índice-cima[i]*

fsi

ffun

fun doble-es-pila-vacía?*(p : pila, i :* 1..2) **dev** *b : bool {©(!))*

si *i* **= 1 entonces** *b := (p.índice-cima[i]* = 0)

si no *b := (p.índice-cima[i] = N* + I)

fsi ffun

3.5. Diseñar una representación dinámica del TAD de las pilas e implementar las operaciones especificadas en el Ejercicio 3.1.

Solución:

Una pila puede representarse mediante una estructura lineal enlazada, de forma que la cima corresponda al extremo directamente accesible de la estructura, con lo que se consigue que el coste en tiempo de todas las operaciones sea constante.

La Figura 3.3 muestra, utilizando dicha representación, una pila de enteros donde se han apilado, sucesivamente, los números —5, 2 y 3.

El tipo representante se define de la siguiente manera:

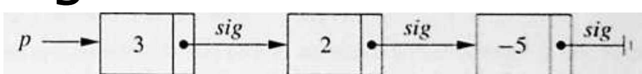


Figura 3.3: Estructura enlazada para una

pila.

tipos

enlace-pila **= puntero a** *nodo-pila*

nodo-pila = **reg**

valor : elemento

sig : enlace-pila

freg

pila = enlace-pila

ftipos

La pila vacía corresponde al caso en el que el enlace exterior no apunta a ninguna estructura.

fun pila-vacía() dev *p : pila* { 9(1) } *p* := nil

ffun

En la implementación de las operaciones de apilar y desapilar se utiliza un enlace auxiliar para hacer la reserva o liberar la memoria dinámica.

proc apilar(e *e : elemento, p : pila)* (9(1))

var <7 : *enlace-pila*

reservarte/)

<7 f *.valor := e : q* f *.sig := p*

p := </

fproc

proc desapilar(/> : *pila)* (9(1) |

var <7 : *enlace-pila*

si *p* = nil **entonces** error(Pila vacía)

si no

'I := P : P := P t *sig*

liberar^/)

fsi

fproc

fun cima(/? : *pila)* **dev** *e : elemento* (9(1) }

si *p* — nil **entonces** error(Pila vacía)
si no *e* := *p* t .*valor*
fsi
ffun
fun es-pila-vacía?(*p : pila)* dev *b : bool*
{ 9(1)) *b* := (/> = nil)
ffun

3.6. Usando la representación dinámica del Ejercicio 3.5. programar una función copiar-pila que dada **una** pila *p* devuelva un puntero a una estructura que represente la misma pila que *p.* pero ocupando posiciones de memoria diferentes; y un procedimiento anular-pila cuyo efecto sea liberar todo el espacio de memoria ocupado por la pila *p* antes de la llamada, dejando como nuevo valor de *p* la pila vacía.

Solución:

Puesto que la representación dinámica de una pila consiste en una estructura lineal enlazada, remitimos al lector a la solución dada para el Ejercicio 2.8 donde precisamente se describen sendos algoritmos para copiar y anular una estructura lineal enlazada. Tan solo se necesitaría cambiar nombres de los algoritmos y los tipos de los parámetros para que sean de tipo *pila.*

Diseñar una representación de las pilas cuando los elementos son dígitos, utilizando números naturales, **e** implementar las operaciones especificadas en el Ejercicio 3.1.

(a) Considerar dígitos entre 1 y 9.

(b) Considerar dígitos entre 0 y 9.

Solución:

Apartado (a)

La idea es utilizar números naturales representados en base 10. de forma que el dígito más significativo (situado más a la izquierda) corresponde al fondo de la pila, y el dígito menos significativo (situado más a la derecha) corresponde a la cima. Por tanto, el tipo representante es el siguiente:

tipos

pila-d1 = nat

ftipos

Obviamente, no todo número natural representa una pila, ya que en este primer apartado no se admiten dígitos 0 en las pilas. La implementación de las operaciones se detalla a continuación.

En pi imer lugar, la pila vacía se representa con el número cero.

fun pila-vacía() dev *p : pila-d1* { 0(1) }
P := 0
ffun

Para apilar un nuevo dígito debemos añadirlo por la derecha del número que representa la pila, es decir, sumamos el dígito al número multiplicado por 10.

proc apilar(e *d* : 1..9, *p : pila-d1)* (0(1))
p := p * 10 + *d*
fproc

Para desapilar debemos eliminar el dígito menos significativo, es decir, dividimos por 10 el número que representa la pila; el resto de dicha división corresponde a la cima de la pila.

proc desapilar(p : *pila-d1)*(0(1))
si *p* **= 0 entonces** error(Pila vacía)
si no *p :*= *p* div 10
fsi
fproc

fun cima(p : *pila-di)* **dev** *d :* 1..9 (0(1) }
si *p* **= 0 entonces** error(Pila vacía)
si no *d :*= *p* mod 10
fsi
ffun

fun es-pila-vacía?(p : *pila-d1)* dev *b : bool* { 0(1) }
b := (p = 0) **ffun**

Suponiendo que las operaciones aritméticas sobre naturales tengan un coste constante, el coste en tiempo de

todas las operaciones también es constante.

Apartado (b)---------------------

El problema al admitir el dígito 0 en las pilas es que el número 0 representa tanto la pila vacía corno una pila con uno o varios ceros. En realidad, tenemos problemas para representar cualquier pila en cuyo fondo haya uno o más dígitos 0. puesto que estos corresponderían a ceros a la izquierda, que no se consideran en la representación decimal de los naturales. Por esta razón, representaremos cada pila con un registro con dos números naturales: el primero indica el número de dígitos 0 en el fondo de la pila, y el segundo representa el resto de los dígitos en la pila. Así. el tipo representante es el siguiente:

tipos
pila-d2 = **reg**
ceros-fondo : nat
dígitos : nat
freg
ftipos

La implementación de las operaciones es la siguiente, donde hay que tener cuidado con los ceros en el fondo de la pila. Por lo demás, las operaciones siguen las ideas de la implementación vista en el apartado anterior.

fun pila-vacía() **dev** *p : pila-d2 {©(!))*
p .ceros-fondo := 0
p. dígitos **:= 0**
ffun
proc apilarte *d :* 0..9, *p : pila-d2)* {

0(1) }

si *p.dígitos* = 0 **entonces** (en la pila solo hay ceros }

si *d* = 0 **entonces** (se añade un cero al fondo)

p.ceros-fondo := p.ceros-fondo + 1

si no (se añade el primer dígito que no es cero)

p.dígitos := d

fsi

si no (se añade el dígito por la derecha)

*p.dígitos := p.dígitos** 10 + *d*

fsi

fproc

proc desapilar(p : *pila-d2'i* {0(1)}

si *p.dígitos* = 0 **entonces (** en la pila solo hay ceros }

si *p.ceros-fondo* **= 0 entonces** error(Pila vacía)

si no (se elimina un cero del fondo }

p.ceros-fondo := p.ceros-fondo — 1

fsi

si no (se elimina el dígito más a la derecha)

p.dígitos p.dígitos div 10

fsi

fproc

fun cima(/>: *pila-d2)* **dcv** *d* : 0..9 { 0(1))

si *p.dígitos* = O **entonces** { en la pila solo hay ceros }

si *p.ceros-fondo* = **0 entonces** error(Pila vacía)

si no (el dígito de la cima es un cero)

¿ := 0

fsi

si no { se devuelve el dígito más a la derecha)

d := p.dígitos mod 10

fsi

ffun

fun es-pila-vacía?(p : *pila-d2)* dev *b : bool* { 0(1))

b := (p.dígitos = 0) A *(p.ceros-fondo* = 0)

ffun

Como en el apartado anterior, si suponemos que las operaciones aritméticas sobre naturales tienen un coste constante, el coste en tiempo de todas las operaciones es constante también.

3.8. Tenemos como dato un vector D[1..TV] que contiene registros, cada uno de los cuales consta de tres campos con información sobre grandes desastres:

- fecha del desastre.
- número de víctimas, y
- la fecha del último desastre anterior que tuvo más víctimas que el presente.

Los registros del vector están ordenados cronológicamente, del más antiguo al más reciente, pero el tercer campo está vacío en todos los registros. El objetivo es completar el vector, rellenando el tercer campo de cada registro a partir de la información que contiene el vector.

(a) Escribir un algoritmo que no utilice

ninguna estructura auxiliar y tenga un coste en tiempo en *@(N²).*

(b) Escribir un algoritmo que utilice una *pilo* como estructura auxiliar y tenga un coste en tiempo en *&(N).*

Solución:

El tipo registro indicado en el enunciado para representar un desastre es como sigue:

tipos
desastre **= reg**
fecha : fecha
víctimas : nat⁺
último : fecha
freg
ftipos

Vamos a representar con el valor especial sm-fecha de tipo *fecha* el hecho de que no hay ningún desastre anterior con más víctimas; en particular, esto es siempre cierto para el primer desastre.

Apartado (a)

Dado que el vector ya está ordenado por fechas, basta recorrerlo desde cada posición *i* hacia la posición 1, para encontrar el último desastre con más víctimas. Si no existe tal desastre, se llegará a la "posición” 0.

proc completar! (£>[!..A] **de** *desastre)* *{ @(N²))*
para *i* = I **hasta** *N* **hacer**
J := ' - 1
mientras *j* > 0 Ar *D[j].víctimas* < *D[i].víctimas* **hacer**
J := J - 1

fmientras
si y' = 0 **entonces** *D[i].último :=* sin-fecha
si no *D[i].ültimo := D[j].fecha*
fsi
fpara
fproc

El coste en tiempo de este algoritmo con dos bucles anidados está en C-)(rV²). tal y como se pedía en el enunciado.

Apartado (b)

Para reducir el tiempo de ejecución, se recurre a la utilización de una estructura auxiliar. En concreto, se utiliza una pila, como sugiere el enunciado, de forma que en la misma queden almacenados solamente los desastres anteriores con un número de víctimas mayor que el último desastre considerado. Los desastres se apilan de tal forma que quedan ordenados cronológicamente, y forman una sucesión decreciente en el número de víctimas, estando el menor en la cima. El desastre de turno se compara con los desastres almacenados en la pila, empezando por la cima (que contiene el desastre en la posición anterior en el vector), hasta encontrar un desastre con un número de víctimas mayor. Los desastres con un número de víctimas menor o igual pueden descartarse definitivamente, ya que no pueden ser el último desastre con más

víctimas para ninguna de las posiciones del vector que quedan por considerar. Si la pila se queda vacía, es que no hay ningún desastre anterior con más víctimas. En cualquier caso, se apila el desastre en estudio para así abordar la siguiente iteración Obsérvese que dicha inserción mantiene la propiedad invariante de que los elementos en la pila forman una sucesión decreciente.

proc completar2(£>[1. JV| **de** *desastre) { ©(?/)* | **var** *p : pila[desastre]*
  p := pila-vacia()
  para *i* = I **hasta** *N* **hacer**
    mientras ~'es-pila-vacía?(p) A r *cima(p).víctimas < D[i].víctimas*
    hacer
      desapilar(*p)*
    fmientras
    si es-pila-vacía?(p) **entonces** *I)[i\.último* := sin-fecha
    sino *D[i\.último :=* cima(*p)fecha*
    fsi
    apilar(D[í], *p)*
  fpara
  anular-pila(p)
fproc

Nótese el uso de anular-pila para liberar la memoria ocupada por la estructura auxiliar al finalizar el proceso de completar el vector.

El coste en tiempo de este algoritmo es lineal con respecto al número de desastres, ya que cada desastre entra

en la pila exactamente una vez. y solo se puede desapilar una vez.

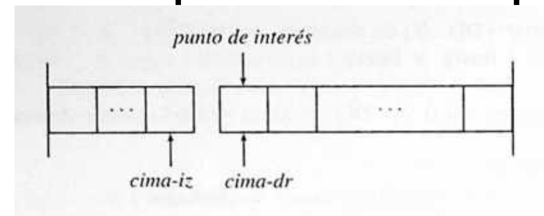


Figura 3.4: Secuencia vista como dos pilas enfrentadas.

3.9. Una *secuencia* es una estructura lineal con un punto de interés donde se realizan las modificaciones y las consultas..Especificar el tipo de las secuencias representando una secuencia con un par de pilas, incluyendo las siguientes operaciones:

- crear una secuencia vacía,
- insertar un elemento delante del punto de interés,
- eliminar el elemento en el punto de interés.
- consultar el elemento en el punto de interés,
- avanzar en un elemento el punto de interés,
- trasladar el punto de interés al comienzo de la secuencia,
- determinar si el punto de interés se encuentra al final de la secuencia, y
- determinar si la secuencia es vacía.

Solución:

Como indica el enunciado, representamos una secuencia mediante un par de pilas: la primera confien la parte de la secuencia a la

izquierda del punto de interés, y la segunda contiene el punto de interés y I parte de la secuencia a la derecha de este. La Figura 3.4 ilustra esta idea. Conviene el uso de estructura LIFO. porque siempre trabajaremos en el mismo extremo de cada parte de la secuencia (la zona d interés), y que será la cima de la pila correspondiente.

Tenemos, por tanto, una operación constructora libre que construye una secuencia a partir de de pilas. Sin embargo, como esta operación constructora es parte de la representación, tiene un cierto caracú "privado”, de forma que conviene que el usuario de las secuencias desconozca su existencia, pudiend utilizar solo las operaciones enumeradas en el enunciado, las cuales estarán definidas en términos de esl constructora.

Algunas de las operaciones son parciales. En concreto, no se puede avanzar, ni eliminar, ni consult; el elemento actual en el punto de interés si estamos al final de la secuencia.

Como no se exige ningún requisito especial para los elementos de una secuencia, consideramos I forma más simple de parámetro.

especificación *SECUENCIAS[ELEMi*
usa *PILAS[ELEM], BOOLEANOS*
tipos *secuencia*

operaciones

crear : —> *secuencia*

insertar : *secuencia*

elemento —>secuencia

eliminar :*secuencia -—>„*

secuencia

actual *secuencia* P *elemento*

avanzar :*secuencia*—►*secuencia*

reiniciar*:secuencia —»* *secuencia*
fin? *:secuencia —>* *bool*
es-sec-vacía? : *secuencia* —> *bool*

operaciones privadas

(: pila pila —> secuencia (constructora)

variables

e : elemento
s : secuencia
iz, dr : pita

Al crear una secuencia no existe ningún elemento en el punto de interés, y ambas partes, izquierda y derecha, están vacías.

ecuaciones

crear = (pila-vacía, pila-vacía)

Al insertar un elemento (por delante del punto de interés) lo colocamos en la parte izquierda: el punto de interés no cambia.

insertar((*iz. dr). e)* = (apilarte. *iz). dr)*

Para eliminar el elemento en el punto de interés tenemos que quitar el primer elemento en la parte derecha, es decir, el elemento en la cima de la segunda pila.

eliminarf*{iz.* pila-vacía)) *= error*
eliminar!(*iz.* apilarte, *dr))) = {iz. dr)*

El elemento en el punto de interés se encuentra en la cima de la parte derecha.

actual((*iz.* pila-vacía)) = error
actual((*iz.* apilar(e, *dr))) = e*

Al avanzar en la secuencia pasamos un elemento de la parte derecha a la parte

izquierda.

avanzartt *iz.* pila-vacía)) = error

avanzar!(*iz.* apilarte, *dr)))* = (apilarte, *iz). dr)*

Para remiciar la secuencia, debemos pasar todos los elementos de la parte izquierda hacia la parte derecha.

reiniciartt pila-vacía, *dr))* = (pila-vacía, *dr)*

reiniciart(apilarte, *iz). dr))* = reiniciartt *iz.* apilarte, *dr)))*

Estamos al final de una secuencia si su parte derecha está vacía.

fin?(< *iz. dr))* = es-pila-vacía?*{dr)*

Una secuencia solo es vacía si su parte izquierda y su parte derecha lo son.

es-sec-vacía?((íz,*dr))* — es-pila-vacía?(í;) A es-pila-vacía?(</r)

fespecificación

Desarrollar una implementación dinámica del TAD de las secuencias especificado **en** el **Ejercicio 3.9,** de manera que todas las operaciones sean ejecutables en tiempo constante.

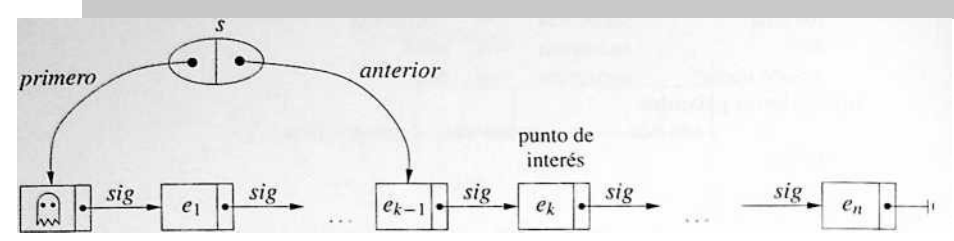


Figura 3.5: Estructura enlazada para una secuencia.

Solución:

La especificación de las secuencias se dio en términos de dos pilas (parte izquierda y parte derecha). Resulta sencillo obtener una implementación de las secuencias mediante pilas, ya sean

implementadas de forma estática (véase el Ejercicio 3.3) o de forma dinámica (véase el Ejercicio 3.5). pero en ese caso el coste en tiempo de la operación reiniciar es lineal con respecto al tamaño de la secuencia, ya que es preciso mover todos los elementos de la pila izquierda hacia la pila derecha. En cambio, la solución que mostramos a continuación, que implementa una secuencia mediante una única estructura lineal enlazada, consigue un coste constante para todas las operaciones de la especificación. Para ello, necesitamos poder acceder en tiempo constante al primer elemento de la secuencia, para poder reiniciar en tiempo constante. También necesitamos acceder en tiempo constante al punto de interés. Como los elementos se insertan *por delante* del punto de interés (véase la especificación en el Ejercicio 3.9). tenemos que acceder también al nodo anterior al punto de interés. Para no tener que distinguir el caso especial en el que el punto de interés está en el primer elemento, resulta conveniente añadir un nodo "fantasma" a la cabecera de la estructura. Dicho nodo no contiene información real alguna, pero con su mera existencia se consigue que todo nodo que contiene un elemento *real* tenga un nodo anterior. Con todo ello, una secuencia vendrá definida por dos punteros, uno que apunta al primer nodo de la estructura (el nodo fantasma) y otro que apunta al nodo anterior al punto de

interés. Dicha representación se ilustra mediante el gráfico de la Figura 3.5. Así la parte izquierda comprende desde el nodo siguiente al fantasma hasta el nodo apuntado por *anterior,* mientras que la parte derecha va desde el nodo siguiente al apuntado por *anterior* hasta el final de la estructura. Con todo esto, el tipo representante es el siguiente:

tipos
enlace-secuencia **= puntero a** *nodo-secuencia*
nodo-secuencia **= reg**
valor : elemento
sig : enlace-secuencia
freg
secuencia **= reg**
primero, anterior : enlace-secuencia
freg
ftipos

Al crear una secuencia vacía solo creamos el nodo fantasma, adonde apuntarán los dos punteros de la secuencia.

fun crear() dev *s : secuencia* { 0(1) j
var *p : enlace-secuencia*
reservar (p)
i.primero := p ; i.anterior := p
p'f.sig := nil
ffun

Para insertar un elemento en la secuencia hay que crear un nuevo nodo, que se enlaza en la estructura por detrás del nodo apuntado por *anterior.* Dicho

nodo pasa a ser el anterior al punto de interés.

proc insertar(s : *secuencia,* **e** *e : elemento)* { 0(1))
var *p : enlace-secuencia*
reservar(p)
p f *.valor := e*
p f *.sig := (i.anterior)* T *.sig*
(i.anterior) f *.sig := p*
i.anterior := p
fproc

Las siguientes operaciones se basan en usar adecuadamente la información de *anterior.* Las tres primeras dan error si nos encontramos al final de la secuencia.

- Para eliminar un elemento se suprime el nodo situado detrás del apuntado por *anterior.*
- Para consultar el elemento actual en la secuencia, se accede al nodo siguiente al apuntado por *anterior.*
- Para avanzar en la secuencia es suficiente trasladar el puntero *anterior* al nodo siguiente.
- Para reiniciar la secuencia basta trasladar el puntero *anterior* al nodo fantasma en la cabecera.
- Estamos al final de una secuencia si no hay nodos tras el nodo apuntado por *anterior.*

proc eliminares : *secuencia)* { 0(1))
var *p : enlace-secuencia*
si *(i.anterior)* f *.sig* — mil **entonces** error(Final de la secuencia)
si no
p := (i.anterior) f *.sig*

(i.anterior) f *.sig := p* f *.sig*
liberar(p)
fsi
fproc
fun actual(s : *secuencia)* **dev** *e : elemento* (0(1))
si *(s.anterior)* f *.sig* = nil **entonces** error(Final de la secuencia)
sino *e := ((i.anterior)* f *.sig)* f *.valor*
fsi
ffun
proc avanzarfs : *secuencia)* j 0(1) }
si *(i.anterior)* f *.sig* = nil **entonces** error(Final de la secuencia)
sino *i.anterior := (i.anterior) .sig*
fsi
fproc
proc reiniciar(s : *secuencia)* (0(1) }
i.anterior := i.primero
fproc
fun fin?(s : *secuencia)* **dev** *b : bool* { 0(1) }
b := ((i.anterior) f *.sig* = nil)
ffun

Finalmente, la secuencia es vacía cuando solo tenemos el nodo fantasma.

fun es-sec-vacía?(.v : *secuencia)* **dev** *b : bool*
b := ((i.primero) f *..sig* = nil)
ffun

Una expresión aritmética construida con los operadores binarios y constantes (representados **cada uno por un carácter)** se dice que está en forma *postfija* si, o bien

es una única constante, o bien consiste en dos expresiones en forma postfija una tras otra, seguidas inmediatamente por un operador. **A continuación** se presenta un ejemplo de una expresión escrita en la notación infija habitual junto con su correspondiente forma postfija:

Formainfija: (A/(B — C)) * (£> + £)

Forma postfija: *ABC — /DE* + *

Diseñar un algoritmo iterativo que calcule el valor de una expresión dada en forma postfija. Se supone que la expresión viene representada como una secuencia de caracteres (véase el Ejercicio 3.9), y que hay disponibles una función valor que asocia a cada constante su valor numérico (real) y una función aplicar que realiza la operación aritmética indicada sobre dos números reales dados.

------------------Solución-----------------

Para evaluar una expresión en forma postfija, vamos a utilizar como estructura auxiliar una pila (de números reales) donde iremos almacenando el valor de los operandos de la expresión y de los resultados intermedios. Se comienza el proceso con una pila vacía y se va recorriendo la secuencia (que contiene la expresión) de izquierda a derecha. Cada vez que se pasa por una constante, se apila su valor. Cada vez que se pasa por un operador, se desapilan los dos valores más arriba en la pila, se componen con el

operador (teniendo en cuenta el orden), y se apila el resultado. Al acabar el proceso la pila contendrá un solo número, que es el valor de la expresión. Aunque el contenido de la secuencia dada como argumento no se modifica, el punto de interés va cambiando al hacer el recorrido, por lo que implcmentamos el algoritmo mediante un procedimiento.

proc evaluar(exp : *secuencia[car], r : real)*

var *p : pilafieal]*

p := pila-vacía()

reiniciar(exp)

mientras --fin?(exp) **hacer**

x := actual (exp) : avanzar(exp)

si es-constante?(x) **entonces**

apilar(valor(x), *p)*

si no (operador)

0P2 := cima(p) ; desapilar(p)

op| := cima(p) ; desapilar(p)

r := aplicar(x, *opi. opfi) :* apilar(r, *p)*

fsi

fmientras

r := cima(p) ; desapilar(p)

fproc

Implementar una función que compruebe si una sucesión de paréntesis abiertos y cerrados está "equilibrada , es decir, a cada paréntesis abierto le corresponde uno cerrado, estando bien anidados. Por ejemplo, ((() ()) ()) está equilibrada, pero (() (no lo está. Se supone que la sucesión viene dada en una secuencia

de caracteres (véase el Ejercicio 3.9). Generalizar el ejercicio suponiendo que además de paréntesis, la sucesión contiene corchetes y llaves que también deberán estar equilibrados. Así, {([))}() está equilibrada, pero ({] { no lo está.

-------------------Solución---------------

En el caso en el que solo haya paréntesis abiertos y cerrados, basta con recorrer la secuencia de caracteres manteniendo en un contador *cont* el número de paréntesis abiertos aún no cerrados. Si en la posición actual de la secuencia se encuentra un paréntesis abierto, incrementamos *cont:* si encontramos un paréntesis cerrado y *cont* > 0. restamos uno al contador y continuamos. Pero si encontramos un paréntesis cerrado y *cont* = 0. entonces la secuencia no está equilibrada. Además, al terminar de recorrer la secuencia, el contador debe ser cero para que la misma esté equilibrada. El coste en tiempo de este algoritmo es lineal con respecto al tamaño de la secuencia.

proc equilibrada?! (s : *secuencia[car], b : bool)*

cont := 0 ; *b* := cierto ; reiniciar(s)

mientras -'fin?(s) A *b* **hacer**

x := actual(s) ; avanzar(y)

si x = (entonces *cont := cont* **+ l**

si no { x =))

si *cont* > 0 **entonces** *cont := cont* —

l

si no *b :*= falso
fsi
fsi
fmientras
b := b A *(cont* = 0)
fproc

Cuando además de paréntesis tenemos llaves y corchetes, necesitamos una estructura auxiliar para guardar los separadores abiertos todavía no cerrados, en el orden en el que han aparecido en la secuencia. La estructura auxiliar que necesitamos es una pila, ya que los separadores se cierran en sentido inverso al que se abrieron Cuando en la secuencia encontramos un separador abierto, este se apila; cuando encontramos uno cerrado, comprobamos si es la pareja de la cima de la pila, siempre que esta no sea vacía. Si la pila es vacía o los separadores no forman pareja, la secuencia de entrada no está equilibrada. Además, al terminar de recorrer la secuencia, la pila debe quedar vacía para que la misma sea equilibrada.

proc equílibrada?2(.v . *secuencia[cari, b : bool)*
var *p : pila[car]*
b := cierto ; reiniciar(s) ; *p* := pila-vacía()
mientras ->fin?(s) A *b* **hacer**
x := actual(.r) ; avanzar(s)
si es-abierto?(x) entonces apilaría, p)
si no { x es cerrado }

```
      si es-pila-vacía?(p) entonces b := falso
      si no
        y ;= cima(p) : desapilar(p)
        b := es-pareja?(x, y)
      fsi
    fsi
  fmientras
  b :== b A es-pila-vacía?(p)
  anular-pila(p)
fproc
```

Nótese que si la secuencia está equilibrada, la pila auxiliar quedará vacía al final del proceso, pero si no está equilibrada, pueden quedar caracteres en la pila, por lo que es oportuno anular la pila auxiliar.

A pesar de la necesidad de manejar la estructura auxiliar, el coste en tiempo de este algoritmo también es lineal con respecto al tamaño de la secuencia, como en el caso anterior.

Capitulo 4

4. COLAS

Probablemente el ciudadano medio (que no las personas influyentes o famosas) pase más de media vida esperando en colas (la cola en la parada del autobús, la cola delante de la ventanilla del banco, la cola para ser atendido en la carnicería, etc.), donde supuestamente *el primero en entrar es el primero en salir.*

En este capítulo vamos a considerar una estructura de datos lineal cuya característica principal es que el acceso a los elementos se realiza en el mismo orden en que fueron almacenados. Aunque se las puede denominar estructuras *F1FO (First In, First Out* en inglés), el nombre más usual, que es el que vamos a utilizar, es *colas.* Este comportamiento de las colas es totalmente independiente del tipo de los datos almacenados en ellas, por lo que se trata de un tipo de datos parametrizado.

A diferencia de la estructura de pila (vista en el capítulo anterior) que tiene un único punto de acceso, las colas presentan dos zonas de interés: el extremo final, por donde se incorporan los elementos, y la cabecera, por donde se consultan y se eliminan los elementos. Las representaciones son un poco más complejas que para las pilas, pero siguen

resultando eficientes.

El comportamiento FIFO es muy utilizado en el diseño de algoritmos para diversas aplicaciones, sobre todo en simulación, debido a la ubicuidad de las colas en toda clase de sistemas.

EJERCICIOS RESUELTOS.

4.1. Especificar un TAD para describir las *colas* con elementos pertenecientes a **un tipo dado como pará**metro. con las siguientes operaciones:

- crear la cola vacía,
- añadir un elemento al final de la cola,
- eliminar el primer elemento en la cola.
- consultar el primer elemento, y
- determinar si la cola es vacía.

Solución:

Puesto que el comportamiento de una cola es independiente del tipo de los datos almacenados en ella, el tipo parámetro no requiere ninguna característica especial, por lo que consideraremos el parámetro más simple: *ELEM.* definido en la Sección 1.1.5.

La elección de constructoras no ofrece muchas alternativas: para obtener una cola se parle de una estructura vacía (cola-vacía) y se van añadiendo los elementos por el final (pedir-vez). Como el orden de inserción en la estructura es fundamental para la posterior consulta y eliminación, las constructoras son *libres.*

Las operaciones de consulta (primero) y eliminación (avanzar) del elemento en la cabecera de una cola solo tienen sentido si la cola no es vacía, por lo que ambas operaciones son parciales.

especificación *COLAS[ELEM]*
usa *BOOLEANOS*
tipos *cola*

operaciones

cola-vacía : —> *cola* { constructora)

pedir-vez *: cola elemento —>*

cola { constructora)

avanzar *:* *cola* ■—*cola*

primero *:* *cola* —*elemento*

es-cola-vacía? :*cola* —> *bool*

variables

e : elemento

c : cola

El comportamiento de la operación modificadora avanzar y de las observadoras primero y es-cola- vacía? viene dado mediante la distinción de casos de las dos constructoras. En realidad, en el caso de primero y avanzar, se distinguen tres casos: no hay elementos en la cola (error), hay un único elemento en la cola (este es el primero) y hay más de un elemento en la cola (hay que acceder al primero de la cola mediante una llamada recursiva). Las ecuaciones correspondientes simplemente reflejan el comportamiento FIFO de una cola: la cabecera corresponde al primer elemento insertado.

ecuaciones

avanzar(cola-vacía) = error

avanzar(pedir-vez(c. e)) = cola-vacía <= es-cola-vacía?(c)

avanzar(pedir-vez(c, e)) = pedir-vez(avanzar(c). *e)* <= ->es-cola-vacia?(c)

primero(cola-vacía) = error

primero(pedir-vez(c. <?)) = *e* <= es-

cola-vacía? (c)

primero(pedir-vez(c, *e))* = primero(c)
<= -■es-cola-vacía? (c)

es-cola-vacía?(cola-vacía) = cierto

es-cola-vacía?(pedir-vez(c, <?)) =
falso

fespecificación

4.2.Extender la especificación de las colas del Ejercicio 4.1 con las siguientes operaciones:

- consultar el último elemento de una cola,
- obtener la inversa de una cola, y
- concatenar dos colas, colocando los elementos de la segunda a continuación de los de la primera.

Solución:

Como en el caso de la operación primero, no tiene sentido consultar el último elemento de una cola si esta es vacía, por lo que la operación último es parcial.

La especificación del comportamiento de estas nuevas operaciones se hace por distinción de casos sobre las dos constructoras de las colas (cola-vacía y pedir-vez).

especificación *COLAS+[ELEM]*
usa *COLAS[ELEM]*
operaciones
último *: cola —elemento*
inversa *cola —> cola*
concatenar *cola cola —> cola*
variables
e : elemento
c.d : cola

El último elemento de una cola es el último que fue añadido a la misma.

ecuaciones
último(cola-vacía) = error
último(pedir-vez(c, c)) = *e*

Para obtener la inversa de una cola no vacía se añade (por el extremo final) el

primer elemento al resultado de invertir la cola resultante de eliminar este.

inversa(cola-vacía) = cola-vacía

inversa(c) = pedir-vez(inversa(avanzar(c)), pnmero(c)) <= -’es-cola-vacía?(c)

Para especificar la concatenación distinguimos casos sobre la *segunda* cola a concatenar: si es vacía, nos quedamos con la primera cola, pero si no lo es, tendremos que añadir el último elemento de la segunda cola a la concatenación de toda la primera cola con el resto de la segunda.

concatenarte. cola-vacía) = *c*

concatenarte. pedir-vez(</. e)) = pedir-vez(concatenar(c. *di. e}*

fespecificación

4.3. Diseñar una implementación estática del TAD de las colas (con las operaciones especificadas en el Ejercicio 4.1) basada en un vector.

Solución:

La idea es utilizar un vector *contenido* donde vamos colocando los elementos de la cola, de izquierda a derecha según se van añadiendo, y dos índices que apuntarán al comienzo *(primero)* y al final *(último}.* Gestionamos el vector de forma "circular", para no tener que desplazar los elementos cuando el vector esté ocupado hasta la última posición pero haya posiciones libres al principio del mismo. Para cada posición en el vector se define su siguiente mediante la función sig:

.... I i + 1 si l < i < N
SI9(,)=| I siÍ = ,V

donde N es el tamaño del vector utilizado.

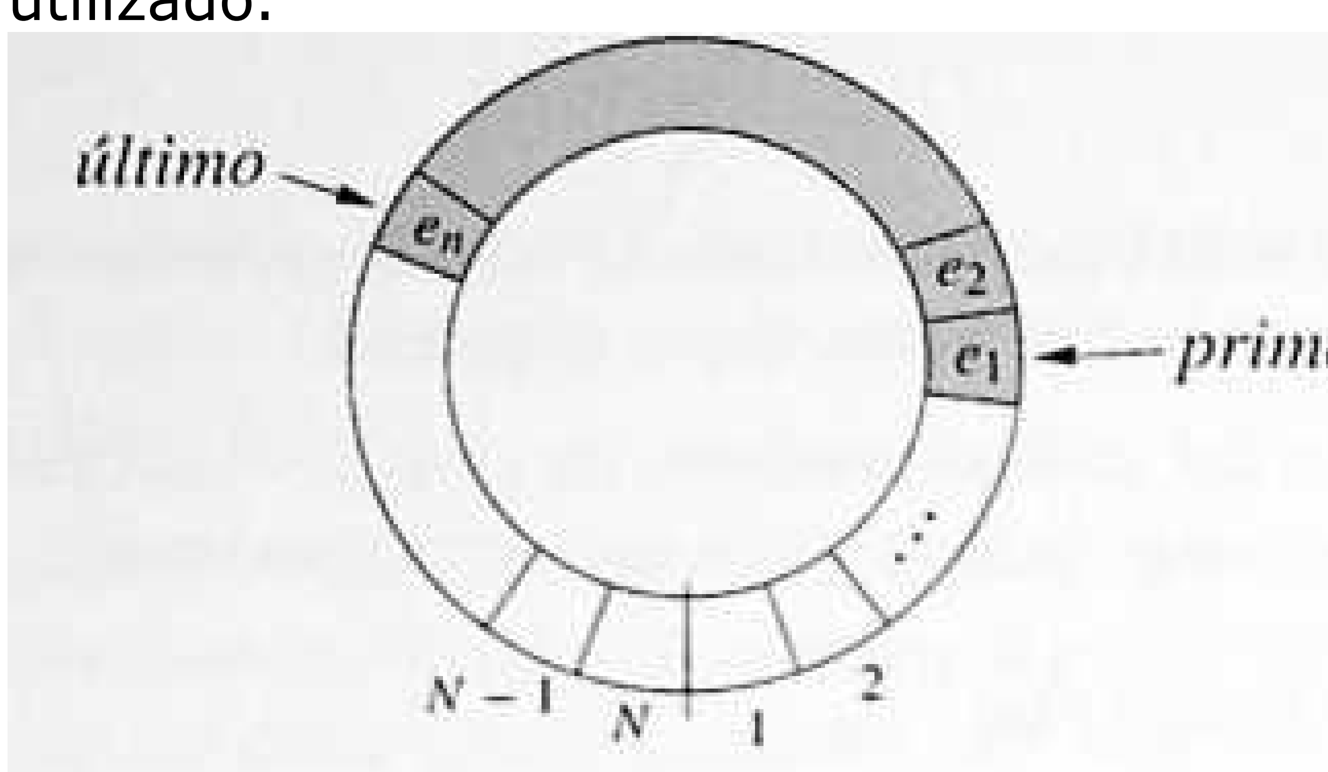


Figura 4.1: Representación circular de una cola.

El dibujo de la Figura 4. i representa el estado general de una cola representada por un vector circular, donde la parte sombreada corresponde a la zona ocupada por los elementos introducidos en la cola.

En el caso vacío, como todavía no hay ni primero ni último, hay que decidir cuáles pueden ser los valores apropiados de los dos índices. Para que la operación pedir-vez no necesite tratar por separado el caso en que se añade por primera vez un elemento a la cola vacía, conviene que *último* sea igual a *N* para que al pasar al siguiente ya valga 1, y *primero* sea igual a I para que se quede apuntando al primero.

Nótese que entonces *sig(último') = primero* en dos ocasiones totalmente diferentes: cuando la cola está vacía y cuando la cola está completamente llena. Para poder distinguir ambas situaciones, necesitamos añadir al registro también el número de elementos en la cola. Otra posibilidad, que no desarrollaremos aquí, es no dejar que el vector se llene por completo, con lo que no se necesitaría llevar el número de elementos de la cola.

El tipo representante es el siguiente:

tipos

cola = reg

contenido[l ..N] **de** *elemento*

primero, último : \..N

tamaño : 0..N

freg

ftipos

La función sig puede implementarse de forma sencilla de la siguiente manera

fun sig(í : 1..7V) dev s : 1../V { 0(1))

s := (í mod *N)* + 1 **fun**

Como ya se comentó en la Sección 2.1. estas representaciones sobre memoria estática tienen el inconveniente de que el tamaño de las estructuras queda restringido por la capacidad del soporte de almacenamiento. en este caso, por el tamaño del vector *contenido.* De esta forma, la operación de añadir resulta ser parcial, produciéndose un error cuando se intenta rebasar dicha capacidad.

La cola se inicializa con *tamaño* igual a cero, señalando *primero* la primera posición del vector y *último* la última.

fun cola-vacía() dev *c : cola* (0(1))

c.tamaño := O

c.printero := 1 ; *c.último := N*

ffun

Una vez se ha comprobado que hay hueco para un nuevo elemento, este se añade en la posición siguiente al último elemento en la cola, y el tamaño se incrementa en uno.

proc pedir-vez(c : *cola,* **e** *e : elemento)* { 0(1))

si *c.tamaño = N* **entonces** error(Espacio insuficiente)

si no

c.último := sig(c.*último)*
c.contenidofc.último] := e
c.tamaño c.tamaño + I
fsi
fproc

Una vez se ha comprobado que la cola no es vacía, se avanza haciendo que *primero* señale a la siguiente posición del primer elemento actual, y el tamaño se decrementa en uno.

proc avanzar(<? : *cola)* (0(1) }
si *c.tamaño* = O **entonces** error(Cola vacía)
si no *c.primero := sig(c.primero) ; c.tamaño := c.tamaño* — **1**
fsi
fproc

El primer elemento es el valor en la posición *primero* de *contenido.*

fun primero(c : *cola)* **dev** *e : elemento* **{ ©(1) |**
si *c.tamaño* = O **entonces** error(Cola vacía)
si no *e c.contenido[c.primero]*
fsi
ffun

La cola está vacía si el tamaño es cero.

fun es-cola-vacía?(c : *cola)* **dev** *b : bool* (0(1))
/> .= (c.tamaño = 0)
ffun

4.4. Diseñar una representación dinámica del TAD de las colas e implementar las operaciones especificadas en el Ejercicio 4.1.

Solución:

Una cola puede representarse mediante una estructura lineal enlazada, que mantenga el orden en que los elementos fueron introducidos en la cola. Teniendo acceso directo a ambos extremos de la estructura se consigue que el coste en tiempo de todas las operaciones sea constante. Dicha representación corresponde al gráfico de la Figura 4.2, donde se utilizan punteros para señalar al primer elemento (extremo por donde se consultan y eliminan los elementos) y al último (extremo por donde se introducen los elementos).

El tipo representante se define de la siguiente manera:

tipos
enlace-cola **= puntero a** *nodo-cola*
nodo-cola **= reg**
valor : elemento
sig : enlace-cola
freg
cola **= reg**
primem. último : enlace-cola
freg
ftipos

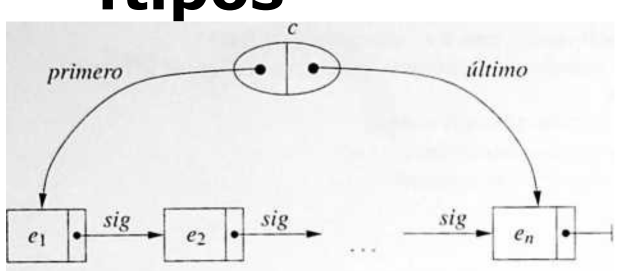


Figura 4.2: Estructura enlazada para una cola.

La cola vacía corresponde al caso en el que los enlaces a los extremos no apuntan a ninguna estructura.

fun cola-vaciaí) **dcv** c : *cola* { 0(1) }
c.primero nil : *c.último* := nil
ffun

En la implementación de la operación de añadir se utiliza un enlace auxiliar para hacer la reserva de la memoria dinámica. Como el elemento se introduce por el final de la cola, el nuevo nodo se enlaza con el último. Solo en el caso de que la cola estuviera vacía es necesario modificar el enlace *primero.*

proc pedir-vez(c : *cola,* **e** *e : elemento)* { 0(1)) **var** *p : enlace-cola*
reservar! p)
pf.valor := e : p^ .sig := nil
si *c.primero* = nil **entonces** *c.primero := p*
si no *(c.último)* j *.sig := p*
fsi
c.último := p
fproc

En la implementación de la operación de avanzar en la cola también se utiliza un enlace auxiliar, en este caso para liberar la memoria dinámica asignada al primer elemento (cuando existe). En el caso de que la cola se quede vacía, es también necesario anular el enlace *último.*

proc avanzar(c : *cola)* { 0(1))
var *p : enlace-cola*
si *c.primero* = nil **entonces**
error(Cola vacía)
si no
p := c.primero ; c.primero := p f
.sig
si *c.primero* = nil **entonces** *c.último*
:= nil **fsi**
liberar(p)
fsi
fproc
fun primero(c : *cola)* dev *e : elemento*
{ 0(1))
si *c.primero* = nil **entonces**
error(Cola vacía)
sino *e := (c.primero)* t *.valor*
fsi
ffun

Para determinar si la cola está vacía, basta comprobar si el enlace al primer elemento es nulo (el enlace al último lo será también).

fun es-cola-vacía?(c : *cola)* **dev** *b : bool*
{ 0(1) } *b := (c.primero* = nil)
ffun

Usando la representación dinámica del Ejercicio 4.4. programar una función copiar-cola que dada una cola *c* devuelva un puntero a una estructura que represente la misma cola que *c.* pero ocupando posiciones de memoria diferentes; y un procedimiento anular-cola cuyo efecto sea liberar todo el espacio de memoria ocupado por la cola

c antes de la llamada, dejando como nuevo valor de *c* la cola vacía.

Solución:

Puesto que la representación dinámica de una cola consiste fundamentalmente en una estructura lineal enlazada, adaptaremos adecuadamente la solución dada para el Ejercicio 2.8 donde se describen sendos algoritmos para copiar y anular una estructura lineal enlazada.

fun copiar-cola(c : *cola)* **dev** *d : cota* **var** r. .s, *t : enlace-cola*
  si *c.priinero* = mi entonces (c es vacía)
    d.primero := nil ; *d.último :*= nil
  si no
    r := c.primero
    reservar(r)
    i j *.valor :*= copiar-elem(r f *.valor)*
    s := i
    d.primero '.= t
    mientras *r* f *.sig* ± nil **hacer**
      r := r f *.sig*
      reservar(r)
      i f *.valor :*= copiar-elem(/- f *.valor)*
      s t *.sig := t*
      s := t
    fmientras
    s f *.sig :*= nil *d .último '.= s*
  fsi
ffun
proc anular-cola(c : *cola)*
var *p. q : enlace-cola*
  p := c.primero
  mientras *p* ± nil **hacer**
    q ■- *p-, p* := pt-síg

anular-elem(<7 f *.valor)*
liberarle/)
fmientras
c.primero := nil ; *c.último* := nil
fproc

(a) Implementar el TAD de las colas (Ejercicio 4.1) utilizando como tipo representante un par de pilas.
(b) Implementar el TAD de las pilas (Ejercicio 3.1) utilizando como tipo representante un par de colas.

-------------------Solución------------------

Apartado (a)

Teniendo en cuenta que una cola tiene dos puntos de acceso (elementos primero y último), mientras que una pila solo tiene uno (cima), es necesaria una segunda estructura para simular el comportamiento FIFO. Como se pide en el enunciado, esta estructura auxiliar será otra pila.

Lo más sencillo es ir apilando en una de las pilas los elementos que se vayan añadiendo a la cola. **De** esta forma, en la cima de la pila se encuentra el último elemento de la cola, mientras que el primero está en el fondo de la pila. En consecuencia, para poder realizar las operaciones de primero y avanzar es necesario acceder al fondo de la pila, y la segunda pila se utiliza como almacén auxiliar. Sin embargo, si traspasamos todos los elementos desde la primera pila hasta la segunda, en esta última se quedará en la cima el primer elemento de

la cola y en el fondo el último, es decir, estarán justamente en el orden que nos interesa para acceder a los elementos de la cola utilizando las operaciones de la pila.

De esta forma, se considera la primera pila de “entrada" y la segunda pila de “salida”:

tipos

cola = reg

entran, salen : pila

freg

ftipos

La cola se inicializa con las dos pilas vacías.

fun cola-vacía() dev *c : cola* (0(1))

c.entran := pila-vacía()

c.salen := pila-vacía()

ffun

Los elementos se añaden a la cola apilándolos en la pila *entran*

proc pedir-vez(c : *cola,* **e** *e : elemento)* j 0(1))

apilar(e, *c.entran)*

fproc

El primer elemento de la cola se encuentra en lacinia de la pila *salen* siempre que esta no esté vacía. En consecuencia, para eliminar el primero de la cola, se desapila en la pila *salen* (y para consultar el primero, se consulta la cima); pero si en ese momento dicha pila está vacía, se moverán todos los elementos desde la pila *entran* a la pila *salen,* quedando en orden inverso a como llegaron. Esta es

la razón por la cual primero se implementa como procedimiento en vez de como función.

proc avanzarte ; *cola)*
si es-pila-vacía? *(c.salen)* **entonces**
si es-pila-vacía?*(c.entran)* **entonces**
error(Cola vacía)
si no
mover-elementos(c)
desapilar(c.safen)
fsi
si no desapilar(c.saZeti)
fsi
fproc
proc primero(c : *cola, e : elemento)*
si es-pila-vacía?(c.sa/en) **entonces**
si es-pila-vacía?*(c.entran)* **entonces**
error(Cola vacía)
si no
mover-elementos(c)
e := cima(c.sa/e/i)
fsi
si no *e* := cima(c.so/<?/i)
fsi
fproc

El procedimiento para mover elementos de la pila *entran* a la pila *salen* se implementa de la forma siguiente:

proc mover-elementos*(c : cola)*
mientras --es-pila-vacía? *(c.entran)*
hacer
e := c\ma(c.entran) :
desapilarfc.entran)
apilarte, *c.salen)*
fmientras

fproc

La cola está vacía solo si las dos pilas lo están.

fun es-cola-vaeía?(c : *cola)* dev *b : bool*

b := es-pila-vacía? (ceníran) A es-pila-vacía?(c.s«/e/t)

ffun

El coste en tiempo de las operaciones cola-vacía y pedir-vez es constante, pero el de las operaciones avanzar y primero es lineal con respecto al número de elementos almacenados en la cola en el caso peor (cuando hace falta moverlos de una pila a la otra). Sin embargo, estos costes se refieren a cada operación de forma independiente. Si consideramos el uso de una estructura dentro de una aplicación, se realizará una secuencia de operaciones sobre dicha estructura. Una opción para calcular el coste total de dicha secuencia sería simplemente sumar los costes individuales de cada operación. Ahora bien, esta aproximación puede resultar excesivamente grosera en determinados casos, al haberse calculado el coste de cada operación individual independientemente de las restantes. En el caso que nos ocupa, si se añaden *n* elementos a una cola representada por dos pilas, para a continuación eliminarlos todos, el coste total no es cuadrático como podría parecer utilizando el análisis en el caso peor, sino lineal, pues la eliminación del

primer elemento requerirá el traspaso de los *n* elementos desde la pila *entran* hacia la pila *salen,* pero el resto de las eliminaciones ya no requerirán ningún traspaso, y cada una tendrá coste constante. De este modo, el coste elevado de unas pocas operaciones de la secuencia se "compensa" con el coste reducido de las demás, y se puede afirmar que el coste promedio de cada operación se mantiene en 0(l). Esta forma de calcular costes se denomina *amortizada, y* para más información sobre este tema remitimos al lector al Capítulo 17 de |CLRS0l].

Apartado (b)-----------------

Aunque una cola tiene dos puntos de acceso y una pila solamente uno. las restricciones de acceso a la cola obligan a utilizar una estructura auxiliar para guardar temporalmente los elementos cuando se desea acceder al último de la cola. Como indica el enunciado, esta estructura auxiliar puede ser otra cola. En el apartado anterior, el movimiento de los elementos de una pila a la otra facilitaba posteriores accesos a la cola representada; pero como las colas conservan el orden de entrada, ahora no se obtiene beneficio alguno, en general, del traspaso de elementos de una cola a la otra. Sin embargo, como al traspasar los elementos de la cola principal a la auxiliar estos quedan en el mismo orden podemos intercambiar el papel de las

colas, en vez de volver a mover todos los elementos de nuevo a la cola principal. Así. el tipo representante estará formado por un vector con dos colas y un índice *i* perteneciente al tipo 1..2 que indica cuál es en cada momento la cola que tiene los elementos. La otra cola (en la posición 3 — *i* del vector) estará vacía hasta que los elementos se muevan a ella al tener que desapilar o consultar la cima de la pila.

tipos

pila = reg

almacén[\..2] de *cola*

i : 1..2

freg

ftipos

La pila se inicializa con las dos colas vacías, y se escoge la primera como almacén actual.

fun pila-vacíaí) dev *p : pila* { 0(1) }

p.almacén[\] := cola-vacía()

p.ahnacén[2] := cola-vacía()

i := 1

ffun

Los elementos se apilan añadiéndolos a la cola que indica *i.*

proc apilar(e *e : elemento, p : pila)* { 0(1))

pedir-vez(p.a/;wícén[í], *e)*

fproc

La cima de la pila corresponde al último elemento de la cola *almacén[i* |. Para llegar hasta el mismo, se va avanzando por la cola *alntacén[i\.* pero guardando los elementos eliminados en la cola *almacén)! — /],* para no

perderlos. Este movimiento necesario de elementos es la razón de que la operación cima se implemente como procedimiento y no como función.

Utilizaremos una operación auxiliar último, que mueve de una cola a la otra todos los elementos, excepto el último, el cual se elimina de la primera cola y se devuelve su valor. Además al final se intercambia el papel de las colas, asignando a *i* el valor 3 — *i.*

proc desapilar(p : *pila)*
si *es-co\a-vacía?(p.almacén[i])*
entonces error(Pila vacía)
si no último(p, *e)*
fsi
fproc
proc cima(/) : *pila, e : elemento)*
si es-cola-vacía?(p.a/nmc<?n[í])
entonces error(Pila vacía)
si no
último(p. *e)*
pedir-vez(p.almacén[i], e) (el elemento *e* no puede perderse }
fsi
fproc
proc último(p : *pila, e : elemento)*
e := primero(a//nace/i[í])
avanzar(a/m«cén(í])
mientras -'es-cola-vacía?(«/m«cé7i[/']) **hacer**
pedir-vez(a/;nacén[3 — /]. *e)*
e := primero(a/znflcói[í]) ;
avanzar(o/inocén[í])
fmientras
i := 3 - *i*

fproc

La pila está vacía si la cola indicada por *i* lo está (la otra cola siempre lo está).

fun es-pila-vacía?*(p : pila)* dev *h : bool* (0(1) i

b := es-cola-vacía?(p.«Z/nacénp])

ffun

El coste de las operaciones pila-vacía y apilar es constante, pero el de las operaciones desapilar y cima es siempre lineal respecto al número de elementos en la pila porque siempre hacen una llamada al procedimiento auxiliar último que mueve todos los elementos de una cola a otra. Un análisis amortizado no mejoraría esta cota.

Considerar una operación para ***entremezclar*** los elementos de 2 colas, de forma que el primer elemento de la cola resultante sea el primer elemento de la primera cola, y a continuación se vayan alternando los elementos de ambas colas.

(a) Especificar la operación entremezclar.
(b) Implementarla de forma abstracta, es decir, sin acceder a la representación interna de las colas.
(c) Implementarla utilizando directamente la representación dinámica de las colas (Ejercicio 4.4).

-----------------Solución----------------

Apartado (a)

Al entremezclar las colas en la forma indicada en el enunciado, ocurrirá que si las colas no tienen la misma longitud, los últimos elementos en la cola resultante corresponderán a los de la cola más larga. Por eso. el último elemento añadido a la cola resultado será el último elemento de la cola de mayor longitud; y si ambas colas fueran de igual longitud, será el último elemento de la segunda cola.

Especificamos también la operación longitud que contabiliza el número de elementos en una cola, y que consideramos operación privada.

especificación *COLAS-ENTREMEZCLAR[ELEM]*

usa *COLAS[ELEM]. NATURALES*

operaciones

entremezclar *cola cola —> cola*

operaciones privadas

longitud : *cola —> nat*

variables

e : elemento

ecuaciones

c, d : cola

longitud(cola-vacía) = 0

longitud(pedir-vez(c, *e))* = 1+1 I + longitud(c)

entremezclar(cola-vacía. cola-vacía) leía) = cola-vacia

entremezclar(pedir-vez(c, *e). d)*) = pedir-vez(entremezclar(c. *d). e)*

<= longitud(c) + I > longitud(</)

entremezclarte, pedir-vez(<7. *e))*) = pedir-vez(entremezclar(c. </). e)

<= longitud(c) < longitud(cZ) + 1

fespecificación

Apartado (b)---------------------

Para la especificación de la operación entremezclar hemos recurrido a la operación longitud; pero la implementación de esta última requiere un coste en tiempo lineal en el número de elementos de la cola correspondiente, por lo que una implementación de entremezclar basada en calcular longitudes resultaría muy costosa.

Como alternativa, presentamos un algoritmo iterativo que avanza por ambas colas simultáneamente, copiando el elemento correspondiente de la primera cola, seguido del de la segunda, hasta que alguna de las colas quede vacía. Para terminar, se copian los elementos sobrantes de la cola que no haya quedado vacía.

El algoritmo se presenta como una

función que a partir de dos colas crea una nueva cola resultado. Para evitar la destrucción de los argumentos, estos se copian en estructuras auxiliares.

fun entremezclar-b(a, *b : cola)* **dev** *c : cola* **var** C], C2 : *cola*
c ;= cola-vacíaQ
<?1 := copiar-cola(n) ; cj := copiar-cola(¿>)
mientras -’es-cola-vacía?(ci) A -’es-cola-vacía?(c2) hacer
e primero(ci) ; avanzar(ci) ; pedir-vez(c, *e)*
e := primero^) ; avanzar^) ; pedir-vez(c, *e)*
fmientras
mientras ~’es-cola-vacía?(<?i) hacer
e := primero(ci) ; avanzar(ci) ; pedir-vez(c. *e)*
fmientras
mientras -’es-cola-vacía?(c2) hacer
e := primero^) ; avanzar(c2) ; pedir-vez(c, *e)*
fmientras
ffun

Si asumirnos que las operaciones básicas de las colas tienen coste en tiempo constante, el algoritmo anterior tendrá un coste lineal con respecto al número total de elementos de las dos colas.

Apartado (c)-----------------

Si “descendemos” al nivel de la representación interna de las colas, en concreto al de su representación dinámica, el algoritmo anterior se puede

expresar en términos de manejo de enlaces, en lugar de llamadas a procedimientos y funciones. En este caso no hace falta copiar los argumentos, porque podemos recorrer las estructuras enlazadas sin destruirlas.

Para facilitar la inserción de los elementos en la estructura enlazada sin tener que distinguir si se trata del primer elemento, creamos un nodo inicial (cabecera) sin valor, que será eliminado al finalizar el proceso.

fun entremezclar-c(a, *b : cola)* **dev** c : *cola*
var *p. p\, p2 : enlace-cola*
resenar(c.primero) ; c.último :=
c.prímero { crear cabecera)
pi := *a.primero ; p2 := b.primero*
mientras *pi* nil A *p2* / nil **hacer**
{ copiar elemento de la primera cola
)
reservar((c.ú/rúno) f *.sig) ; c.último*
:= (.c.último) f *.sig*
(c.último) t *.valor :=* pj -f *.valor*
(copiar elemento de la segunda cola
)
reservar! *(c.último)* f *.sig) ; c.último*
:= (c.último) f *.sig*
(c.último) t *.valor := p2* f *.valor ;*
(c.último) f *.sig :=* nil

(avanzar en ambas colas }
P\ := Pi'T-J'g: *P2 := Pi'l-sig*
fmientras
{ copiar elementos restantes de la primera cola |
mientras *p*\ nil **hacer**
reservarle.último) f *.sig) ; c.último := (c.último)* -f *.sig*
(c.último) \ .valor := pi f *.valor ; (c.último)] .sig* := nil Pi *:= p* t *-sig*
fmientras { copiar elementos restantes de la segunda cola) **mientras** *p2* nil **hacer**
reservar(*(c.último)* f *.sig) ; c.último := (c.último)* f *.sig*
(c.último) T *.valor := p2* t *.valor ; (c.último)* f *.sig* := nil *P2* ■= *P*\ t *-sig*
fmientras { eliminar cabecera) *p := c.primero c.primero := (¿.primero)* | *.sig* liberar(p)
ffun

El coste en tiempo sigue siendo lineal respecto al número total de elementos en las dos colas, pero el coste en espacio es mejor porque pasa de ser lineal a ser constante.

(a) Especificar un tipo de datos *doble-cola* de colas dobles que permiten operaciones para consultar, añadir y eliminar elementos en cualquiera de los dos extremos de la estructura lineal.

(b) Realizar una implementación dinámica para la cual todas las operaciones del **tipo tengan un coste** en tiempo

constante.

(c) Implementar operaciones para copiar una doble cola en posiciones de **memoria diferentes y para** liberar el espacio ocupado por una doble cola.

------------------Solución--------------------

Apartado (a)

especificación *DOBLECOLAS[ELEM]*
usa *BOOLEANOS*
tipos *doble-cola*
operaciones
dc-vacía —* *doble-cola*
poner-detrás : *doble-cola elemento—>doble-cola*
poner-delante : *elemento doble-cola —>doble-cola*
quitar-últ : *doble-cola —doble-cola*
último *doble-cola —► elemento*
quitar-prim : *doble-cola —doble-cola*
primero : *doble-cola —elemento*
es-dc-vacía?: *doble-cola —> bool*
variables
e, f : elemento c : doble-cola

A semejanza de las colas ordinarias (Ejercicio 4.1). las operaciones de consulta y eliminación están indefinidas en el caso vacío. Como primer paso tenemos que decidir qué operaciones se toman como constructoras. La posibilidad que vamos a seguir en esta solución es considerar como tales dc-vacía y poner-detrás (si se considera poner-delante en vez de poner-detrás se tiene el caso simétrico), que se corresponden con las constructoras para las colas sencillas cola-vacía y pedir-vez. respectivamente,

y que constituyen un conjunto de constructoras *libres.* De esta forma, las ecuaciones correspondientes a las operaciones quitar-prim (véase avanzar), primero y es-dc-vacía son semejantes a las dadas para las colas sencillas. En cuanto a las operaciones quitar-últ y último, su especificación es muy sencilla porque la constructora introduce los elementos precisamente por detrás, por lo que en el caso no vacío son las ''destructoras" asociadas a poner-detrás. Por último, consideremos poner-delante: en una cola vacía es indiferente poner por delante o por detrás; pero si la cola no está vacía, para poner el elemento delante, habrá que "empujarlo" hacia el principio desde el extremo final.

ecuaciones

poner-delante(e. dc-vacía) = poner-detrás(dc-vacía. e)

poner-delante(e. poner-detrás(c. /)) = poner-detrás(poner-delante(<?. c), *f)*

quitar-últ(dc-vacía) = error

quitar-últ(poner-detrás(c. e)) = c

último(dc-vacía) = error

último(poner-detrás(c, *e))* = *e*

quitar-prim(dc-vacía) = error

quitar-prim(poner-detrás(c, <?)) = dc-vacía <= es-dc-vacía?(c)

quitar-prim(poner-detrás(c. <?)) = poner-detrás(quitar-prim(c). <?) <= -■es-dc-vacía? (c)

primero(dc-vacía) = error

primero(poner-detrás(c, <?)) = *e* <= es-dc-vacía?(c)

primero(poner-detrás(c. *e))* = primero(c) <= -■es-dc-vacía?(c)
es-dc-vacía? (dc-vacía) = cierto
es-dc-vacía?(poner-detrás(c. <?)) = falso

fespecificación

Apartado (b)

A la hora de implementar las colas dobles por medio de una estructura lineal enlazada, resulta insuficiente mantener los enlaces a ambos extremos (como se propone para las colas sencillas en la solución del Ejercicio 4.4). puesto que en tal caso, para poder eliminar el último nodo, nos veríamos obligados a recorrer toda la estructura desde el primer nodo hasta localizar el enlace al penúltimo nodo, con el consiguiente coste lineal para esta operación. Por ello se propone la utilización de una estructura *doblemente enlazada,* donde cada nodo incoipora enlaces tanto al nodo siguiente como al anterior. Con ello podemos obtener procedimientos ejecutables en tiempo constante para todas las operaciones del tipo. La Figura 4.3 muestra gráficamente dicha representación. El tipo representante es el siguiente:

tipos
enlace-doble-cola nodo-doble-
doble-cola

puntero a *nodo-doble-cola*
reg
valor: elemento
sig. ant : enlace-doble-cola
freg

reg
primero, último : enlace-doble-cola
freg
ftipos

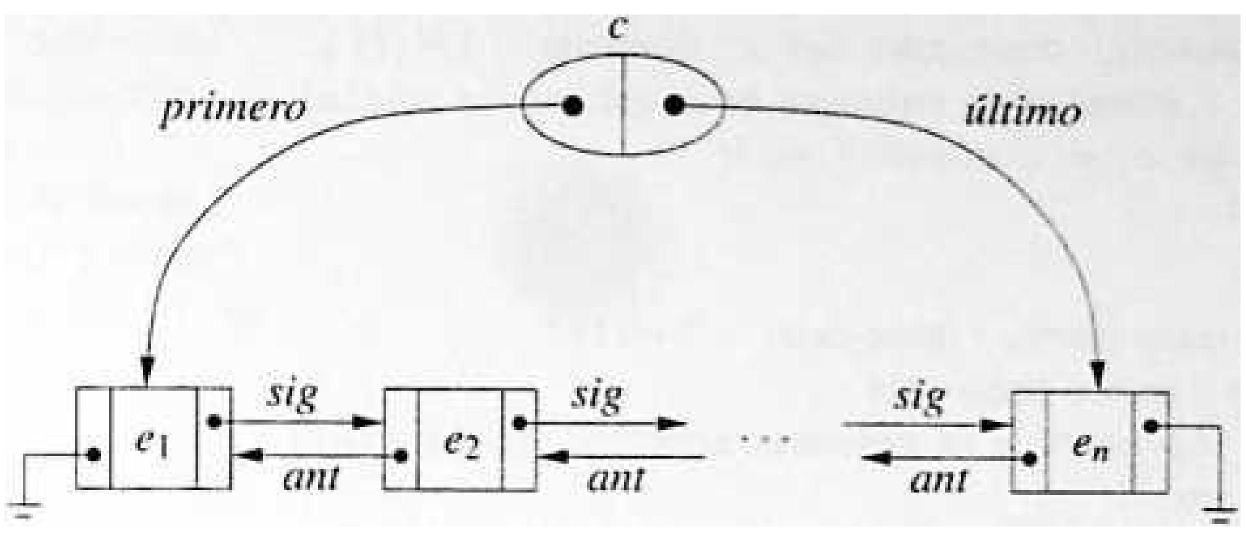


Figura 4.3: Estructura doblemente enlazada para una cola doble.

La implementación de las operaciones que se corresponden con las de las colas sencillas es prácticamente la misma dada en la solución del Ejercicio 4.4. salvo que ahora es necesario tener en cuenta el doble enlace. En cuanto a las operaciones adicionales, su implementación es dual a la de las operaciones que trabajan en el extremo contrario; es decir, se obtienen sustituyendo en ellas *primero* por *último y sig* por *ant,* y viceversa.

fun dc-vacíaO **dev** *c : doble-cola* **(0(1)**
|
c.primero := nil : *c.último* := nil
ffun
proc poner-detrás(c : *doble-cola,* **c** *e : elemento)* (0(1))
var *p : enlace-doble-cola*
reservare *p)*
p],valor := e; *p* f *.sig* := nil
si *c.último* = nil **entonces** *p* f *.ant* := nil

; *c.primero := p*
  sino *p^ .ant := c.último ;* (c. *último)* f *.sig := p*
  fsi
  c.último := p
fproe
proc poner-delante(e *e : elemento, c : doble-cola)* (0(1)) **var** *p : enlace-doble-cola*
  reservare/?)
  p f *.valor — e : p* t *.ant* := nil
  si *c.primero* = nil **entonces** *pt.sig :=* nil: *c.último p*
  sino *p* f *..sig := c.primero. (c.primero)* f *.ant := p*
  fsi
  c.primero := p
fproe
proc quitar-últ(c : *doble-cola)* { 0(1))
var *p : enlace-doble-cola*
  si *c.último* = nil **entonces** errorfDoblecola vacía)
  si no
    p := c.último', c.último := p^.ant
    si *c.último* = nil **entonces** *c.primero :=* nil
    si no *(c.último)* f ..sig := nil
    fsi
    liberar(p)
  fsi
fproe

fun último(c : *doble-cola)* **dev** *e : elemento* { 0(1) }
  si *c.último* = nil **entonces** error(Doblecola vacía)
  si no *e := (c.último)* f *.valor*
  fsi
ffun
proc quitar-prim(c : *doble-cola)* { 0(1))
var *p : enlace-doble-cola*
  si *c.primero* = nil **entonces** error(Doblecola vacía)
  si no
    p '.= c.primero ; c.primero := p | .sig
    si *c.primero* = nil **entonces** *c.último* := nil
    si no *(c.primero)* f *.ant* := nil
    fsi
    liberar(p)
  fsi
fproc
fun primero(c : *doble-cola)* **dev** *e : elemento* { 0(1) }
  si *c.primero* = nil **entonces** error(Doblecola vacía)
  si no *e := (c.primero)* f *.valor*
  fsi
ffun
fun es-dc-vacía?(c : *doble-cola)* **dev** *b : bool* (0(1) }
  b := (c.primero — nil)
ffun

Apartado (c)----------------

Completamos esta implementación de las colas dobles con los algoritmos correspondientes para copiar y anular,

que son muy similares a los dados en la solución del Ejercicio 4.5. pero teniendo en cuenta el doble enlace.

```
fun copiar-dc(c : doble-cola) dev d :
doble-cola
var r.s, t : enlace-doble-cola
  si c.primero = nil entonces d.primero
:= nil ; d.último := nil
  si no
    r := c.primero
    reservar(r)
    t f .valor := copiar-elem(r f .valor)
    t f .ant := nil
    5 := t
    ¿.primero := t
    mientras r f .sig nil hacer
      r := r'f.sig
      reservar(t)
      t j .valor := copiar-elem(r | .valor)
      t f .ant := 5 ; .s f .sig := t
      s := t
    fmientras
    s f .sig := nil
    ¿.último := 5
  fsi
ffun
proc anular-dc(c : doble-cola)
var p, q : enlace-doble-cola
  p := c.primero
  mientras p nil hacer
    anular-elemf/) f .valor)
    q - p \ p := p f .sig
    liberarte/)
  (mientras
  c.primero := nil ; c.último := nil
fproc
```

Se dice que una frase es *palíndroma* si la sucesión de caracteres obtenida al recorrerla de izquierda a derecha (ignorando los blancos intermedios) es la misma que si el recorrido se hace de derecha a izquierda, como por ejemplo en la frase *dábale arroz a la zorra el abad* (se consideran los caracteres sin tilde). Desarrollar tres funciones iterativas de coste lineal en tiempo, que decidan si una frase dada como sucesión de caracteres (leída desde el teclado) es o no palíndroma, utilizando en cada caso como estructuras auxiliares las siguientes:

(a) dos pilas,
(b) una pila y una cola,
(c) una cola doble (Ejercicio 4.8).

Solución:

Apartado (a)

Esta primera solución consiste en almacenar en una pila todos los caracteres no blancos, a medida que se leen desde el teclado, de forma que quedarán en el orden inverso al de lectura. Al tiempo vamos contando el número de tales caracteres leídos. A continuación, se transfieren la mitad de los caracteres almacenados a una segunda pila, quedando estos en el orden original. Si la frase tiene un número impar de caracteres no nulos, el que ocupa la posición central ha de ser ignorado, al ser simétrico de sí mismo. Por último, se comparan las dos pilas; si son iguales, es que la frase de entrada es palíndroma.

```
proc palindroma?-a()
var p\. P2 : pila[car]
  ( leer desde la entrada hacia la primera
pila }
  pi := pila-vacía()
  cont := 0
  leer(.v)
  mientras v / fin hacer
    si .V # entonces apilarU, p) ; cont :=
cont + 1 fsi
    leer(.v)
  fmientras
  { pasar la mitad de la primera pila a la
segunda |
  P2 := pila-vacía()
  para i = 1 hasta (cont div 2) hacer
    .v := cima(pi); desapilar(/>i)
    apilarfv, pi)
  fpara
  si es-impar?(cont) entonces
desapilar(/>i) fsi
    { comparar las dos pilas, del mismo
tamaño )
    b := cierto
    mientras b A -■es-pila-vacía?(pi)
hacer
      b := (cima(pi) = cima(p2))
      desapilar(pi) ; desapilartp?)
    fmientras
    si b entonces imprimir(Es
palíndroma)
    si no
      imprimir(No es palíndroma)
      anular-pila(pi) ; anular-pila(p2)
    fsi
  fproc
```

Nótese que en el caso de que la frase no sea palíndroma al final se procede a la anulación de las dos pilas auxiliares. En el caso de que la frase sea palíndroma, esto no es necesario, puesto que ambas pilas han quedado vacías.

Se hacen 3 recorridos sucesivos, cada uno de los cuales tiene un coste lineal con respecto al número de caracteres leídos, por lo que el coste total sigue siendo lineal. Como se utilizan dos pilas auxiliares, el coste en espacio también es lineal con respecto al mismo tamaño.

Apartado (b)------------------

En esta segunda solución, la idea es recorrer la sucesión de entrada apilando los caracteres no blancos en una pila (donde quedarán en el orden inverso) y al mismo tiempo en una cola (donde quedarán en el orden de entrada). A continuación, se hace un segundo recorrido comparando los caracteres almacenados en la pila con los almacenados en la cola. La frase es palíndroma cuando todos los caracteres coinciden.

proc palíndroma?-b()
var c : *cola[car], p : pila[car]*
{ leer desde la entrada hacia la cola y la pila)
c := cola-vacía()
p := pila-vacía()
leer(x)
mientras x fin **hacer**
si x **entonces** pedír-vez(c. x) ;

apilar(x, p) **fsi**
leer(x)
fmientras
(comparar la cola con la pila)
b cierto
mientras *b* A -■es-cola-vacía?(c)
hacer
b := (primero(c) = cima(p))
avanzar(c) ; desapilar(p)
fmientras
si *b* **entonces** imprimir(Es palíndroma)
si no
imprimir(No es palíndroma)
anular-cola(c) ; anular-pila(p)
fsi
fproc

Aunque ahora solamente se hacen dos recorridos sucesivos, como cada uno tiene un coste lineal con respecto al número de caracteres leídos, el coste total sigue siendo lineal. Ahora se utilizan una pila y una cola auxiliares, por lo que el coste en espacio sigue siendo lineal con respecto al mismo tamaño.

Apartado (c)------------------

En la tercera solución, la sucesión de entrada se guarda (salvo los blancos) en una cola doble, que permite el acceso por ambos extremos. De esta forma, se irá comparando el primer carácter con el último, eliminándolos de la estructura hasta acabar con todos o encontrar alguna discordancia.

proc palíndroma?-c() var c - *doble-cola[car]*

```
  | leer desde la entrada hacia la cola doble }
  c := dc-vacía()
  leer(.v)
  mientras y fin hacer
    si y r entonces poner-detrás(c. ,v) fsi
    leer(.v)
  fmientras
  { comparar la cola doble desde ambos extremos )
  b := cierto
  mientras b A -'es-dc-vacía?(c) hacer
    /> := (pnmero(c) = último(c))
    quitar-prim(c)
    si -'es-dc-vacia?(c) entonces quitar-últ(c) fsi
  fmientras
  si b entonces imprimir(Es palíndroma)
  si no
    imprimir(No es palíndroma)
    anular-dc(c)
  fsi
fproc
```

Los dos recorridos sucesivos y el uso de una única estructura auxiliar no cambian los costes lineales en tiempo y espacio de los apartados anteriores.

El *agente 0069* ha inventado un nuevo método de codificación de mensajes secretos. El mensaje original *X* se codifica en dos etapas: en primer lugar. *X* se transforma en *X'* reemplazando cada sucesión de caracteres consecutivos que no sean vocales por su inversa. En segundo lugar. X' se transforma en la sucesión de caracteres

X" obtenida al ir lomando sucesivamente el primer carácter de X'. luego el último, luego el segundo, luego el penúltimo, etc. Por ejemplo.

X=Anacleto, agente secreto
X'= Analceto ,agetnes erceto
X"= Aontaelccreet os e.natge

Suponiendo que los mensajes se representan mediante secuencias de caracteres (véase el Ejercicio 3.9). diseñar las funciones de codificación y de decodificación de mensajes.

Solución:

Podemos implementar la primera etapa de la codificación de la siguiente manera: se recorre la secuencia de entrada y se van guardando en una pila los caracteres mientras estos no sean sírcales. Cuando se alcance una vocal, se desapilan los caracteres de la pila, que formarán parte del resultado de este primer paso. Se utiliza una pila como estructura auxiliar porque la codificación exige un tratamiento LIFO de los caracteres. Tras vaciar la pila, se añade al resultado la vocal. Al alcanzar el final de la secuencia dada, no hay que olvidar volcar al resultado los caracteres que pudieran haber quedado apilados. Para facilitar

la segunda etapa de la codificación, el resultado de la primera se va almacenando en una cola doble de caracteres (véase el Ejercicio 4.8). que permite el acceso desde ambos extremos.

Implementamos este primer paso como procedimiento porque, aunque el contenido de la secuencia dada *X* no se modifica, el punto de interés cambia al hacer el recorrido de la secuencia.

proc codificación! *(X : secuencia[car]. X' : doble-cola[car])*
var *p : pila[car]*
  X' := dc-vacía() ; *p* := pila-vacíaO
  reiniciar(X)
  mientras —>fin?(X) **hacer**
    x := actual(X) ; avanzar(X)
    si -'es-vocal?(x) **entonces** apilar(x, /?)
    si no
      volcar-p-dc (/?, X')
      poner-detrás(X'. x)
    fsi
  fmientras
  volcar-p-dc(p. X')
fproc

El procedimiento auxiliar volcar-p-dc añade por detrás a una cola doble todos los elementos de una pila (desde la cima hasta el fondo), dejando a esta última vacía.

proc volcar-p-dc(/; : *pila[car], X : doble-cola[car])*
  mientras -▪es-pila-vacía?^) **hacer**
    x := cima(p) ; desapilar(p)

poner-detrás(X. x)
fmientras
fproc

Nótese que al terminar de ejecutar el algoritmo codificación! la pila auxiliar habrá quedado vacía, por lo que no será necesario anularla.

El coste del procedimiento volcar-p-dc es lineal con respecto al número de elementos en la pila. El procedimiento codificación! recorre una única vez la secuencia de entrada. Cada carácter vocal se apila también una única vez en la pila auxiliar (y se desapila desde volcar-p-dc). Por ello, el coste en tiempo de codificación! es lineal con respecto al número de caracteres en la secuencia de entrada. El coste en espacio también es lineal, por la pila auxiliar utilizada.

El segundo paso de la codificación es sencillo si se recibe como argumento una doble cola de caracteres, ya que basta ir recorriéndola sacando alternativamente elementos del principio y del final, añadiéndolos a la secuencia resultado. Lo implementamos como un procedimiento porque la cola doble que se recibe como argumento queda anulada tras la ejecución de este segundo paso.

proc codificación2(X' : *doble-cola[car], X" : secuencia[car])*
X" := crear()
n := l
mientras -■es-dc-vacía?(X') **hacer**
si impar(/t) **entonces** x := primero(X') ; quitar-prim(X')

si no x := último(X') : quitar-últ(X')
fsi
insertar(X", x)
n := n + l
fmientras
fproc

El coste de codificación2 es lineal respecto al número de caracteres en A" ya que se hace un recorrido completo de esta secuencia, aunque alternando los extremos por donde se consultan los elementos.

El procedimiento completo de codificación es el siguiente:

proc codificación(X : *secuencia[car], X" : secuencia[car])*
var *X' : doble-cola[car]*
codificaciónl *(X. X')*
codificación2(X'. X")
fproc

El coste en tiempo del algoritmo codificación es lineal con respecto al número de caracteres en X por las explicaciones anteriores sobre los costes de codificaciónl y codificación?. El coste en espacio es también lineal por la doble cola y la pila auxiliares utilizadas.

La primera etapa de la decodificación (inversa de la segunda etapa de la codificación) se implementa de la siguiente manera: se recorre la secuencia de entrada y se van añadiendo al resultado los caracteres en posiciones impares, y a una pila auxiliar los caracteres en posiciones pares. Al alcanzar el final de la secuencia de

entrada se añaden al resultado los caracteres de la pila, que saldrán en orden inverso. En este caso, la estructura auxiliar para almacenar el resultado intermedio es una cola sencilla, pues la segunda etapa se limitará a realizar un recorrido en el sentido usual.

```
proc decodificaciónl (X" : secuencia[car], X' : cola[car])
var p . pila[car]
  X := cola-vacía() ; p := pila-vacía()
  reiniciar(X")
  n := 1
  mientras ~'fin?(X") hacer
    v = actual(A'") ; avanzar(X")
    si impar(n) entonces pedir-vezGY', ,v)
    si no apilarf.r. /?)
    fsi
    n := n + 1
  fniientras
  mientras -^es-pila-vacía?(p) hacer
    v = cimaí/i) : desapilarí/?)
    pedir-vez( X'. A )
  fniientras
fproc
```

El coste en tiempo de decodificaciónl es lineal debido al recorrido que se hace de toda la secuencia y a que cada elemento solo se apila y desapila una única vez. El coste en espacio es también lineal, por la pila auxiliar utilizada.

La segunda etapa de la decodificación es similar a la primera etapa de la codificación, ya que al calcular dos veces la inversa en las secuencias entre vocales obtenemos la secuencia inicial. Ahora la

entrada es una cola sencilla de caracteres y la salida es una secuencia de caracteres, pero lo implementamos como procedimiento, porque la cola de entrada terminará anulada.

proc decodificación2(X' : *cola[car], X : secuencia[car])*
var *p : pilafcar]*
X crear() ; *p :*= pila-vacía()
mientras -■es-cola-vacia?(X') **hacer**
,v := primero(zY') ; avanzar(zY')
si -■es-vocal?(,v) **entonces** apilar(.v.
/>)
si no

volcar-p-s (yt. ***X)*** insertare X, _v) **fsi**
fmientras
volcar-p-s(p. X)
fproc

El procedimiento auxiliar volcar-p-s inserta por el final de una secuencia todos los elementos de una pila (desde la cima hasta el fondo), dejando a esta última vacía.

proc volcar-p-s(p : *pila[car], X : secuencia[car])* **mientras** -■es-pila-vacía?(p) **hacer** .v := cima(p) ; desapilar(p) insertar(X, A)
fmientras
fproc

El coste de volcar-p-s es lineal con respecto al número de elementos en la pila El coste en tiempo de decodiflcación2 es también lineal con respecto al número de caracteres en la cola X'; al igual que el coste en espacio, por la pila auxiliar utilizada.

Así, el procedimiento completo de decodificación, con un coste lineal tanto en tiempo como en espacio, es el siguiente:

proc decodificación(X" : *secuencia[car], X : secuenciafcar])* **var X' :** *cola[car]*
decodificaciónl (X". X')
decodificación2(X', X)
fproc

4.11. Una *cola medieval* se comporta como una cola ordinaria, con la única diferencia de que los elementos almacenados en ella se dividen en dos estamentos: *nobles* y *plebeyos.* Dentro

de cada estamento, los elementos deben ser atendidos en orden de llegada: además, siempre que haya nobles en la cola, estos deben ser atendidos antes que los plebeyos.

(a) Especificar un tipo abstracto de datos para las colas medievales, que disponga al menos de las siguientes operaciones:

- crear una cola medieval vacía.
- añadir un nuevo elemento,
- consultar el primer elemento.
- quitar el primer elemento.
- consultar el número de nobles en una cola.
- consultar el número de plebeyos en una cola, y
- consultar si una cola es vacía.

En particular, es necesario especificar adecuadamente los requisitos del parámetro que suministra los elementos sobre los que se definen las colas medievales.

(b) Diseñar una implementación de las colas medievales en términos de colas ordinarias, de forma que las implementaciones de todas las operaciones tengan coste constante, suponiendo asimismo coste constante para las operaciones sobre colas ordinarias.

(c) Implementar de nuevo las operaciones sobre colas medievales, pero esta vez utilizando directamente memoria dinámica.

Solución:

Apartado (a)

Se necesita un parámetro que proporcione un predicado es-noble? sobre los elementos (que denominaremos *personas)* para saber cuáles son nobles. Los que no son nobles se consideran plebeyos.

parámetro *GENTE*
usa *BOOLEANOS*
tipos *persona*
operaciones
es-noble? : *persona —> bool*
fparánietro

Las colas medievales se especifican mediante dos operaciones constructoras: la que crea la cola vacía y la que añade personas a la cola. Estas constructoras *no son libres,* puesto que aunque el orden es importante entre nobles y plebeyos por separado, es indiferente entre nobles y plebeyos mezclados, pues los primeros siempre deberán ser tratados con mayor deferencia. Como ocurre con las colas ordinarias, las operaciones de consulta y de eliminación de la primera persona en la cola medieval son parciales, pues solo tienen sentido cuando la cola no es vacía.

especificación *COLAS-MEDIEVALES[GENTE]*
usa *BOOLEANOS. NATURALES*
tipos *cola-med*
operaciones

[illegible]

variables

c : cola-med
\ . v : persona

ecuaciones

(el orden de entrada es indiferente si las personas son de distinto estamento)

cm-pedir-vez(x. cm-pedir-vez(v, *c))* =
cm-pedir-vez(\. cm-pedir-vez(x. <•))
<= es-noble?(.v) A -■es-noble?(y)

Obsérvese que una ecuación con la condición es-noble?(v) A -•es-noble? (A) no es necesaria, porque sería la misma ecuación leída de derecha a izquierda.

Las operaciones modificadoras y observadoras se definen distinguiendo casos sobre las dos constructoras y. si es necesario, mediante la operación es-noble? (proporcionada por el parámetro) para saber si una persona es noble. En particular, las operaciones que consultan o eliminan el primero de la cola tienen que distinguir varias posibilidades, de forma que si la cola tiene nobles el primero de la cola es el primer noble, mientras que solamente cuando en la cola no hay nobles el primero de la cola es el primer plebeyo.

cm-primero(cm-vacía) = error

cm-primero(cm-pedir-vez(c, x)) = cm-primero(c)<= nnobles(c) 0

cm-primero(cm-pedir-vez(c, x)) = .v <= nnobles(c) = 0 A es-noble?(x)

cm-primero(cm-pedir-vez(c, A*)) = cm-primero(c)
<= nnobles(c) = 0 A -^es-noble? (X) A -•es-cm-

aaaaaaaaaa

vacia?(e)

cm-primero(cm-pedir-vez(x. <■)) = .v

<= nnobles(c) = 0 A -•es-noble?(A) A es-cm-vacia?(c)

cm-avanzar(cm-vacía) = error

cm-avanzar(cm-pedir-vez(c, x)) = cm-pedir-vez(cm-avanzar(c), x)<= nnobles(c) 0

cm-avanzar(cm-pedir-vez(c, x)) — *c* <= nnobles(c) = 0 A es-noble?(x)

cm-avanzar(cm-pedir-vez(c, x)) = cm-pedir-vez(cm-avanzar(c), x) <= nnobles(e) = 0 A -■es-noble?(x) A -■es-cm-vacía?(c)

cm-avanzar(cm-pedir-vez(c, x)) = *c* <= nnobles(c) = 0 A -■es-noble?(x) A es-cm-vacía?(c)

El número de ecuaciones para cada una de estas operaciones se puede reducir de cinco a tres combinando mediante una disyunción las condiciones de los pares de ecuaciones que tienen la misma parle derecha, pero al hacerlo así la presentación resulta más confusa. Por la misma razón, no se han simplificado algunas condiciones.

Las ecuaciones para el resto de las operaciones observadoras son muy simples.

nnobles(cm-vacía) = 0

nnobles(cm-pedir-vez(c, x)) = 1 + nnobles(c) <= es-noble?(x)

nnobles(cm-pedir-vez(c, x)) = nnobles(c) <= -^es-noble? (x)

nplebeyos(cm-vacía) = 0

nplebeyos(cm-pedir-vez(c. x)) = nplebeyos(c) <= es-noble?(x)

nplebeyos(cm-pedir-vez(c. x)) = 1 + nplebeyos(c) <= -■es-noble?(x)

es-cm-vacía?(cm-vacía) = cierto
es-cm-vacía?(cm-pedir-vez(c, x)) = falso
fespecificación
Apartado (b)
En esta primera implementación, una cola medieval se representa mediante un registro que incluye un par de colas ordinarias, una para guardar los nobles y otra para los plebeyos. Además, para poder calcular con un coste constante el número de nobles y el de plebeyos, el registro incluye también dos campos con contadores para esa información. Nótese que la información en estos contadores se podría obtener calculando la longitud de las dos colas, pero de esta forma no se podría garantizar el coste constante de las operaciones pues su coste dependería del de calcular la longitud de una cola ordinaria.
tipos
cola-med = **reg**
nobles : cola[persona]
plebeyos : cola[persona]
imobles : nat
iiplebeyos : nat
freg
ftipos
Inicialmente las dos colas soporte serán vacías, y cada persona que entre en la cola medieval se añadirá a la cola (ordinaria) correspondiente a su estamento.
fun cm-vacía() dev *c : cola-med* **{**

0(1) }

c.nobles := cola-vacía() ; *c.nnobles* := 0

c.plebeyos := cola-vac(a() ; *c.nplebeyos :=* 0

ffun

proc cm-pedir-vez(c : *cola-med,* e x : *persona)*

si es-noble?(x) **entonces** pedir-vez(c.noWes, **x) ;** *c.nnobles := c.nnobles* + 1

si no *ped\r-Mez(c.plebeyos,* **x) ;** *c.nplebeyos := c.nplebeyos* + 1

fsi

fproc

El primero de la cola medieval será el primero en la cola de nobles, si esta no está vacía: en otro caso, será el primero de la cola de los plebeyos. Si ambas colas están vacías, tenemos que la cola medieval está vacía.

fun cm-primero(c : *cola-med)* dev x : *persona (©(!))*

si -^es-cola-vacía?(c.*nobles)* **entonces** x := primero(c.noWes)

si no

si -•es-cola-vacía?*(c.plebeyos)* entonces x *:= pumente.plebeyos)*

si no error(Cola medieval vacía)

fsi

fsi

ffun

proc cm-avanzar(c : *cola-med)* (0(l))

si --es-cola-vacía?(c./ioWes) entonces

avanzarte.*nobles) ; c.nnobles :=* *c.nnobles* — l

si no
si *-•es-cola-vacia? (c.plebeyos)*
entonces
avanzarle.plebeyos) ; c.nplebeyos := c.nplebeyos — l
si no error(Cola medieval vacía)
fsi
fsi
fproc
fun nnobles(c *cola-med)* dev *n : nat* (0(l))
n := c.nnobles **ffun**
fun nplebeyos(c *: cola-med)* **dev** *n : nat {©(!)}*
n c.nplebeyos
ffun
fun es-cm-vacía?(c : *cola-med)* dev *b : bool* (0(l) }
b := es-cola-vacia? (c .nobles A es-cola-vacía?*(c.plebeyos)* **ffun**

Otra posibilidad para comprobar que la cola medieval está vacía es la siguiente:

fun es-cm-vacía?2(c : *cola-med)* **dev** *b : bool* { 0(l) }
b := (c.nnobles **= 0) A** *(c.nplebeyos =* **0) ffun**

Apartado (c)--------------------

Una posible implementación dinámica se basa en considerar la típica implementación dinámica para colas ordinarias (véase el Ejercicio 4.4) y aplicarla sobre la implementación más abstracta descrita en el apartado anterior, de forma que una cola medieval se representaría mediante un

par de estructuras enlazadas, con los enlaces externos apropiados, sustituyéndose en cada uno de los programas anteriores las operaciones sobre colas ordinarias por el código correspondiente que maneja los enlaces de las colas representadas.

En esta solución vamos a describir una implementación que intenta simplificar esta idea, representando una cola medieval mediante una única estructura enlazada donde todos los nobles se encuentran delante de los plebeyos. Se tendrá un registro con enlaces al principio *(primero).* para eliminar (nobles) por delante; al final *(último),* para añadir (plebeyos) por detrás: y en el punto de separación entre nobles y plebeyos *(últ-noble),* para añadir nobles sin tener que pasar por los plebeyos. Como en el apartado anterior. el registro incluirá también la cantidad de nobles y de plebeyos, para poder calcularlos en tiempo

Figura 4.4: Estructura enlazada para una cola medieval.

constante. Una representación gráfica de la cadena de punteros aparece en la Figura 4.4, donde N indica que el elemento es noble y P que es plebeyo.

El tipo representante se define de la siguiente manera:

tipos

enlace-cola-med **= puntero a** *nodo-cola-med*

nodo-cola-med = **reg**

persona : persona

sig : enlace-cola-med

freg

cola-med **= reg**

primero : enlace-cola-med

ólt-noble : enlace-cola-med

último : enlace-cola-med

imobles : nat

itplebeyos : nat

freg

ftipos

Para crear la cola medieval vacía, todos los enlaces se inicializan al valor nulo.

fun cm-vacía(c : *cola-med)* { 0(1))

c.primero := nil ; *c.últ-noble '.*= nil ; *c.último* nil

c.nnobles := 0 ; *c.nplebeyos* := 0

ffun

El siguiente algoritmo añade una persona x a la cola medieval, utilizando un enlace auxiliar *p* para crear el nodo que contiene el valor x que se va a insertar en la cadena de nodos que representa la cola. Vamos a explicar con cierto detalle los distintos casos que se distinguen en el algoritmo, y lo que se hace en cada uno de ellos con los correspondientes enlaces.

- Cuando x es noble, se incrementa el número de nobles y se tiene que insertar el nodo apuntado por *p* entre los nobles y los plebeyos ya existentes, de forma que *p* pasa a apuntar al último noble, indicado por *últ-noble.* La forma en que esta inserción afecta a los otros dos enlaces, *primero* y *último,* depende de si hay o no nobles en la cola, y también de si hay o no plebeyos en la cola. Una forma de comprobar ambas condiciones es utilizar los contadores */mobles* y *//plebeyos,* respectivamente. Otra posibilidad es comparar los propios enlaces: el número de nobles es cero exactamente cuando *últ-noble* — nil y el número de plebeyos es cero exactamente cuando *últ-noble* — *último.* Distinguimos entonces los siguientes casos:

- ■ La cola está vacía (no hay nobles ni plebeyos): el nodo apuntado por *p* pasa a ser el primer y último nodo de la cola.

La cola solamente contiene plebeyos: el nodo apuntado por /> se inserta al principio utilizando el puntero *primero* y pasa a ser el primero, mientras que el último de la cola (algún despreciable plebeyo) no cambia.

- ■ La cola solamente contiene nobles: el nodo apuntado por *p* se inserta al final utilizando el puntero *último,* y pasa a ser el último, mientras que el primero de la cola (algún respetable noble que ya formaba parte de la estructura) no cambia.
- La cola contiene tanto nobles como plebeyos: el nodo apuntado por *p* se inserta a continuación del último noble, mientras que el primero y último no cambian.

En todos los casos *p* se convierte en el último noble.

- Cuando v es plebeyo, se incrementa el número de plebeyos y el nodo apuntado por *p* se inserta siempre al final, pasando a ser el último de la cola, y sin afectar al último noble. Ahora solo hay que distinguir si la cola es vacía o no (mirando, por ejemplo, si *último* = nil) para, en caso afirmativo, hacer que el nodo insertado sea también el primero, porque en esa situación es el único nodo en la estructura.

proc cm-pedir-vez(c : *cola-med.* **e .r :** *persona)* {0(1)} **var** *p : enlace-cola-*

med
reservar) *p)*
p f *.persono :=* v : pf.JÍg := nil
si es-noble?(p) **entonces**
c.nnobles := c.nnobles + 1
casos
c.últ-noble = mi A *c.último = c.últ-noble —>*
c.primero := p : c.último := p
▫ *c.últ-noble* — nil A *c.último* 0 *c.últ-noble -—*
p f *.sig := c.primero ; c.primero := p*
D *c.últ-noble =£* nil A *c.último = c.últ-noble —**
(c.último) f *.sig := p : c.último := p*
▫ *c.últ-noble* nil A *c.último c.últ-noble —>*
p f *.sig (c.últ-noble)* T *.sig ; (c.últ-noble)* T *.sig := p*
(casos
c.últ-noble := p
si no
< .iiplebeyos := c.nplebeyos + I
si *e.último* nil **entonces** *c.último* f *.sig := p*
si no *c.primero := p*
fsi
c último := p
fsi
fproc

La primera persona en la cola corresponde al primer nodo de la estructura.

fun cm-primero(c : *cola-med)* dev ,v :

persona (0(1))
si *c.primero* = nil **entonces** error(Cola medieval vacía)
si no .v *:= (c.primero)* f *.persona*
fsi
ffun

El siguiente algoritmo elimina el primero de la cola, utilizando un enlace auxiliar *p* para apuntar al nodo que se va a quitar de la cadena que representa la cola. Si la cola es vacía se produce un error: en otro caso *p* apunta al nodo que indica *primero,* y el primero pasa a ser su siguiente. Si no quedan más nodos *(primero* = nil) los otros dos enlaces también tienen que anularse. Por otra parte, dependiendo de si la

persona eliminada es noble o plebeya, hay que decrementar el correspondiente contador. Finalmente, si hemos eliminado el último noble *{mobles* = 0), hay que anular el enlace *últ-noble.*

proc cm-avanzar(c : *cola-med)* { ®(1) } **var** *p : enlace-cola-med*
si *c.prímero* = nil **entonces** error(Cola medieval vacía)
si no
p := c.prímero ; c.prímero := {c.prímero) t *.sig*
si *c.prímero* = nil **entonces** *c.último :=* nil ; *c.últ-noble* := nil **fsi**
si es-noble?(p f *.persona)* **entonces**
c.nnobles := c.nnobles — 1
si *c.nnobles* = 0 **entonces** *c.últ-*

noble := nil **fsi**
si no *c.nplebeyos := c.nplebeyos* —
1
fsi
liberar(p)
fsi
fproc

El resto de las operaciones resulta muy sencillo de implementar.

fun nnobles(c : *cola-med)* **dev** *n : nat* { 0(1))
n := c.nnobles
ffun
fun nplebeyos(c : *cola-med)* dev *n : nat* (0(1))
n := c.nplebeyos
ffun
fun es-cm-vacía?(c : *cola-med)* dev *b : bool* (0(1) }
b := {c.prímero = nil)
ffun

Dado un número natural *n > 2.* se llaman *números supervivientes* a los que resultan de ejecutar el **siguiente** proceso: a partir de la secuencia 1.2 .... *n,* se comienza eliminando el primer número de cada dos (es decir, los números 1. 3,5. etc.); a continuación, se elimina el primer número de cada tres; etc. El proceso termina cuando se va a eliminar el primer número de cada *ni* pero "sobreviven" menos de *ni* números. Diseñar un algoritmo que reciba *n* como parámetro y devuelva la cola formada por los números

supervivientes resultantes.

-----------------Solución----------------

La utilización de una cola como estructura soporte permite conservar el orden en la secuencia de números. La cola se inicializa con los *n* primeros naturales y, para facilitar el proceso, se mantiene una variable *long* que indica la longitud de la misma.

{ u > 2)

fun supervivientes!» : *nat*$^{+}$*)* **dev** *c : cola[narí]*

c := cola-vacía()

para *i* = 1 **hasta** *n* **hacer**

pedir-vez(c, *i)*

fpara

long := n (número de elementos en la cola)

m := 2

mientras *in* < *long* **hacer**

filtrar(c, *ni, long)*

m := *ni* + I

fmientras

ffun

Para eliminar de una cola de longitud *long* el primero de cada *ni* elementos, es preciso reiterar *long* veces el siguiente proceso: extraer el primer elemento de la cola y añadirlo al final de la misma, salvo si le toca ser eliminado.

I *ni > 2* y *long* es el número de elementos en *c)*

proc filtrarle : *cola,* **e** *ni* : naf[1]", *long : nat)* { 0(/ong))

long-inicial := *long*

para *i* = 0 **hasta** *long-inicial* — 1 **hacer**

A := pnmero(c)

avanzar(c)

si *i* mod *ni* = 0 **entonces** *long* := *long* — 1

si no pedir-vez(c, A)

fsi

fpara

{ *long* es el número de elementos que quedan en *c)*

fproc

Cada iteración del bucle **mientras** en la función supervivientes es lineal con respecto al número de elementos en la cola, como indica el coste de filtrar. En la primera iteración hay *n* elementos y se quitan la mitad; en la segunda hay " *y* se quitan la tercera parte, por lo que

quedan jt en la tercera hay j y se quitan la cuarta parte, por lo que quedan etc. De esta forma, el coste del bucle es proporcional a la siguiente suma, donde *nio* es el último valor de *ni* en el bucle:

"*io ,*

11 11 11 n **< 1**

7 + - + -+■•■ H = *n* 2 ¯ •

I 23 *m0 j*

En la iteración r-ésima del bucle. *111* vale *i* + 1 y hay " elementos en la cola. Como el bucle termina cuando *ni* es mayor que la longitud de la cola, tenemos que *nio* - 1 < ,7^75 >' > *lü^í-*

de dontle

deducimos que *IIIQ* es aproximadamente Vñ. Por otra parte.

* 1

ln(fc +1) < 22 - < 1 + In

Con todo esto tenemos que el coste del bucle **mientras** está en &(*n* In **s/ñ) = 0(n loga).**

Como el coste del primer bucle es lineal, el coste total de supervivientes está en ®(n logn).

Capitulo 5

5. LISTAS

Vivimos rodeados de listas: la lista de la compra, la lista de regalos de boda, la lista de libros para el próximo curso, etc., siendo el denominador común de todas estas listas el establecer una ordenación entre sus elementos.

Se puede afirmar que las listas son las estructuras de datos lineales más flexibles, puesto que su única característica es imponer un orden entre los elementos almacenados en ellas. Según este criterio, las pilas y las colas (véanse los Capítulos 3 y 4) serían casos particulares de listas, donde se ha restringido la forma de acceso a la estructura. En cualquier caso, no existe una visión unánime de las listas como TAD. con un conjunto básico de operaciones, por lo que. en los ejercicios siguientes, se propondrán especificaciones alternativas con las operaciones más usuales sobre listas.

De nuevo se trata de un tipo de datos parametrizado, donde el comportamiento de las operaciones sobre los valores del tipo *lista* es independiente de la naturaleza de los elementos que la componen. Al margen de las constructoras empleadas en cada caso, emplearemos una notación

general para los valores del tipo *lista:* de esta forma, el término [ei.es e„] representa la lista formada por los elementos
<?i, <ʲ2 ■ hasta í'„. en el orden indicado.

5.1.Especificar un TAD para describir las *listas* con elementos pertenecientes a un tipo dado como parámetro, y las siguientes operaciones:

– crear la lista vacía,
– generar una lista unitaria formada por un elemento dado,
– añadir un elemento por la izquierda,
– añadir un elemento por la derecha.
– consultar el elemento más a la izquierda,
– consultar el elemento más a la derecha.
– eliminar el elemento más a la izquierda,
– eliminar el elemento más a la derecha.
– determinar si una lista es vacía,
– concatenar dos listas, y
– calcular la longitud de una lista.

-------------------------Solución:--------------------------

El comportamiento de las listas es independiente del tipo de sus elementos, por lo que las especificaremos de forma paramétrica. Las operaciones del enunciado no requieren ninguna característica especial del tipo parámetro, por lo que utilizaremos el parámetro *ELEM,* definido en la Sección 1.1.5.

Una primera posibilidad a la hora de elegir constructoras consiste en considerar como tales la lista vacía ([ ]) y la operación que añade un elemento por la izquierda a una lista (_: _). Como el orden de los elementos dentro de la lista es esencial, así como las repeticiones,

este conjunto de constructoras es *libre.* Las operaciones que consultan o eliminan los elementos de los extremos son parciales, estando definidas solo en el caso no vacío.

especificación *LISTAS[ELEM]*
usa *BOOLEANOS, NATURALES*
tipos *lista*
operaciones
[] : —> lista (constructora)
: elemento lista —> lista
{constructora}
[_] : elemento —> lista
_# _ :lista elemento —> lista
izquierdo *: lista —elemento*
elim-izq *: lista —lista*
derecho *:lista —>* $_p$ *elemento*
elim-der *:lista—>*$_p$ *lista*
es-lista-vacía? : *lista —> bool*
_ -H- _ *lista lista —> lista*
longitud *:lista—> nat*
variables
e, f : elemento
x, y, z : *lista*

Construir una lista unitaria es equivalente a añadir (por la izquierda) el elemento en cuestión a la lista vacía. Y añadir un elemento por la derecha a una lista equivale a concatenar dicha lista con la lista unitaria formada con el elemento.

ecuaciones
[e] = *e: [ ]*
x # e = x -H- [e]

El resto de las operaciones se definen distinguiendo casos según las

constructoras. En el caso de las operaciones de consulta y eliminación del elemento más a la derecha se distinguen tres casos: la lista vacía, la lista con un elemento y la lista con al menos dos elementos.

izquierdo([]) = error

izquierdo(e:;r) = *e*

elim-izq([ ]) = error

elim-izq(e :x) = .r

derecho([]) = error
derecho(e : [ ]) = *e*
derecho(e : *f* : x) = derecho(/:x)
elim-der([]) = error
elim-der(e: [ ]) = |]
elim-der(e : *f*: x) = *e* : elim-der(/:x)
es-lista-vacía?([ ]) = cierto
es-lista-vacia?(e: x) = falso
[]++y = *y*
(e : x) 44- y *- e:* (x 4+ y)
longitud(|]) =0
longitudíc x) = 1 + longitud(x)

(especificación

También podemos considerar como constructoras la operación que crea la lista vacía y la que añade un elemento por la derecha. La situación es simétrica a la anterior, por lo que no la desarrollaremos (consúltese el Ejercicio 1.5 sobre cadenas finitas).

Una tercera posibilidad consiste en elegir como constructoras la lista vacía, la operación de construcción de una lista unitaria y la operación de concatenación, ya que cualquier lista no vacía puede construirse concatenando listas unitarias. En este caso las constructoras *no son libres.* Las ecuaciones de equivalencia expresan que la concatenación de listas es asociativa y tiene a la lista vacía como elemento neutro.

ecuaciones

| | 44- x = x (elemento neutro |
v -H- | I = x (elemento neutro)

(x 44- y) 4+z = x 44- (y 44-z) { asociatividad)

Las operaciones que añaden un elemento por la izquierda o por la derecha se definen directamente en términos de las constructoras, sin distinguir casos.

e ■ x = [<?] 44- x

x#e = x 44-[e]

El resto de las operaciones se definen distinguiendo casos sobre las constructoras y mediante el predicado es-lista-vacía? (para encontrar explicaciones más detalladas, consúltese el Ejercicio 1.5 sobre cadenas. que es muy similiar).

izquierdo(x) = error <= es-lista-vacia?(x)

izquierdo([e]) = e

izquierdo(x 44-y) = izquierdo(x) <= -■es-lista-vacía?(x)

izquierdo(x 4-4- y) = izquierdo(y) <= es-lista-vacía?(x) A -’es-lista-vacía?(y)

elim-izq(x) = error <= es-lista-vacia?(x)

elim-izq([e]) = []

ellm-izq(x 44- y) = elim-izq(x) 44-y 4= -’es-lista-vacía?(x)

elim-izq(x 44- y) = elim-izq(y) 4= es-lista-vacía?(x) A -■es-lista-vacía?(y)

derecho(x) = error <= es-lista-vacía?(x)

derecho([e|) = e

derecho(x 44-y) = derecho(y) <= -'es-lista-vacía?(y)

erecho(.v 44- y) = derecho(x) <= es-lista-

vacía?(y) A -'es-lista-vacía?(x)

elim-der(x) = error <= es-lista-vacía?(.r)
elim-der([e]) = [ ]
elim-der(.r 44- y) = ,r 4+ elim-der(y) <= -■es-lista-vacía?(y)
elim-der(.v 44-y) = elim-der(.r) <= es-lista-vacía? (y) A -■es-lista-vacía? (,v)
es-lista-vacía?([ ]) = cierto
es-lista-vacía?([e]) = falso
es-lista-vacía?(,v 44- y) = es-lista-vacía? (A) A es-lista-vacía? (y)
longitud([])— 0
longitud(H) = 1
longitud(A-44-y) = longitud(.r) 4-longitud(y)

[especificación

Extender la especificación de las listas del Ejercicio 5.1 con las siguientes operaciones:

- determinar- si un elemento aparece en una lista,
- localizar la posición de un elemento (devolviendo 0 si el elemento no está),
- calcular el número de repeticiones de un elemento,
- eliminar todas las apariciones de un elemento,
- determinar si dos listas son iguales, y
- determinar si una lista es capicúa.

Utilizar como constructoras:

(a) la lista vacía y añadir por la izquierda,
(b) la lista vacía, lista unitaria y concatenación.

Solución:

Todas las operaciones indicadas en el enunciado requieren que los elementos

con los que se forman las listas tengan una operación de igualdad, por lo que utilizaremos el parámetro *ELEM=* (véase la Sección 1.1.5).

especificación *LISTAS+[ELEM=]*
usa *LISTAS[ELEM=], NATURALES. BOOLEANOS*
operaciones

está? : *elemento lista —> bool*
posición : *elemento lista —> nat*
repeticiones *elemento lista —» nat*
eliminar :*elemento lista —> lista*
_ == _ : *lista lista —> bool*
es-capicúa? : *lista —> bool*

variables

e. f : elemento
x, y : lista

Apartado (a)

Las operaciones se definen distinguiendo casos sobre las constructoras. En el caso de la operación está?, ningún elemento está en la lista vacía, y un elemento está en una lista no vacía si es igual a su elemento más a la izquierda o está en el resto de la lista.

ecuaciones

está?(c, [ ]) = falso
está?(/, *e*: x) = *f == e v está?(f.x)*

Para localizar la posición de un elemento en una lista se distinguen casos según el elemento esté o no en la lista. Si el elemento no está se devuelve 0 (la operación también podría haberse definido como parcial, devolviéndose error en tal caso). Si el elemento está se distingue si está en la primera posición o

en el resto.

posición(<?. x) =0 4= -■está?(e.x)

posición (e. *e* :x) = 1

posición(<?. *f :* x) = I + posiciónfe, x)
e= e / / A está?(e. x)

Para contar las repeticiones de un elemento se distinguen casos por constructoras. En el caso no vacío se distingue también si el elemento más a la izquierda es el que estamos contando o no.

repeticionesfe, [|) =0

repeticiones(e, e :x) = 1 + repeticiones(e, x)

repeticiones(e. *f :x)* = repeticionesfe. x)
<= e f

Los mismos casos se distinguen para eliminar todas las apariciones de un elemento, solo que ahora los elementos no eliminados tienen que permanecer en la lista.

eliminarte. [ ]) = [ ]

eliminarte, e : x) = eliminarte, x)

eliminarte./: v) = *f* eliminarte, x) <= e *f*

La lista vacía solo es igual a ella misma, y dos listas no vacías son iguales si lo son tanto sus elementos más a la izquierda como sus restos. Obsérvese que en la parte derecha de la cuarta ecuación se utiliza el símbolo == para denotar tanto la igualdad entre elementos (primera aparición) como la igualdad entre listas que se está definiendo recursivamente (segunda aparición).

| | == |] = cierto

[ ] == *e* : x = falso

e ■ x == [ ] = falso
e : x == *f y* —*e* == *f* A x == .V
La lista vacía y la lista con solo un elemento son capicúas. Una lista con más de un elemento es capicúa si el elemento más a la izquierda coincide con el de más a la derecha y la lista resultante de eliminar ambos es también capicúa.
es-capicúa^{9}([ ]) = cierto
es-capicúa?(e []) = cierto
es-capicúa?(e:x) = *e* == derecho(x) A es-capicúa?(ehm-der(x)) <= --es-lista-vacía? (x) **(especificación**
Apartado (b)---------------------------
Para definir las operaciones está?, repeticiones y eliminar se distinguen casos según las constructoras. Para el resto de las operaciones no es útil la descomposición que ofrecen las constructoras en este apartado, por lo que se distinguen casos mediante condiciones y se utilizan las "destructoras" izquierdo, derecho, elim-der y elim-izq.
ecuaciones
está?(e, []) = falso
está?(e, [/]) = *e* == *f*
está?(e, x 44-y) = está?(e.x) v está?(e. y)
posición(e, x) = 0 4= -=está?(e. x)
posición(e,x) = 1 <= -■es-lista-vacía?(x) A izquierdo(x) == *e*
posición(e.x) = 1 + posición(<?. elim-izq(x))
4= -'es-lista-vacía?(x) A está?(e. elim-izq(x)) A izquierdo(x) / *e*

repeticiones(e, [ ]) =0
repeticiones^. [<?]) = 1
repeticiones (e. [/]) = 0 <= *e / f*
repeticiones(e. x +1- y) = repeticiones(e, x) 4- repeticiones (e, y)
eliminarte. []) = []
eliminarte, [e]) = [ ]
eliminarte, [/]) = [/] 4= e / *f*
eliminarte, x 4+y) = eliminarte, x) 4+eliminarte, y)
[] = x = es4ista-vacía?(x)
x==[] = es4ista-vacía?(x)
x == y = (izquierdo(x) == izquierdo(y)) A es4gual?(elim4zq(x), elim-izq(v))
<= ~'es4ista-vacía?(x) A -’es-lista-vacía(y)
es-capicúa?(x) = cierto <= longitud(x) < 1
es-capicúa?(x) = (izquierdofx) == derecho(x)) A es-capicúa?(elim-izq(elim-der(x)))
4= longitud (x) > 1

fespecificación

Diseñar una representación dinámica del TAD de las listas e implementar las operaciones del Ejercicio 5.1.

--------------------Solución---------------

Los elementos de una lista se representan mediante una sucesión de nodos doblemente enlazados: cada nodo, además del valor del elemento concreto almacenado, contiene enlaces al nodo siguiente y al anterior en la sucesión. La ventaja de tener estos dobles enlaces es que así se pueden eliminar elementos por ambos extremos de la lista con un coste

en tiempo constante. Una lista será un registro con un campo que guarda la longitud de la lista y enlaces a los nodos más a la izquierda y más a la derecha de la sucesión. Dicha representación se ilustra en la Figura 5.1.

El tipo representante es el siguiente:

tipos
enlace-lista = **puntero a** *nodo-lista*
nodo-lista **= reg**
valor: elemento
sig, ant : enlace-lista
freg
lista **= reg**
longitud : nat
izquierdo, derecho : enlace-lista
freg
ftipos

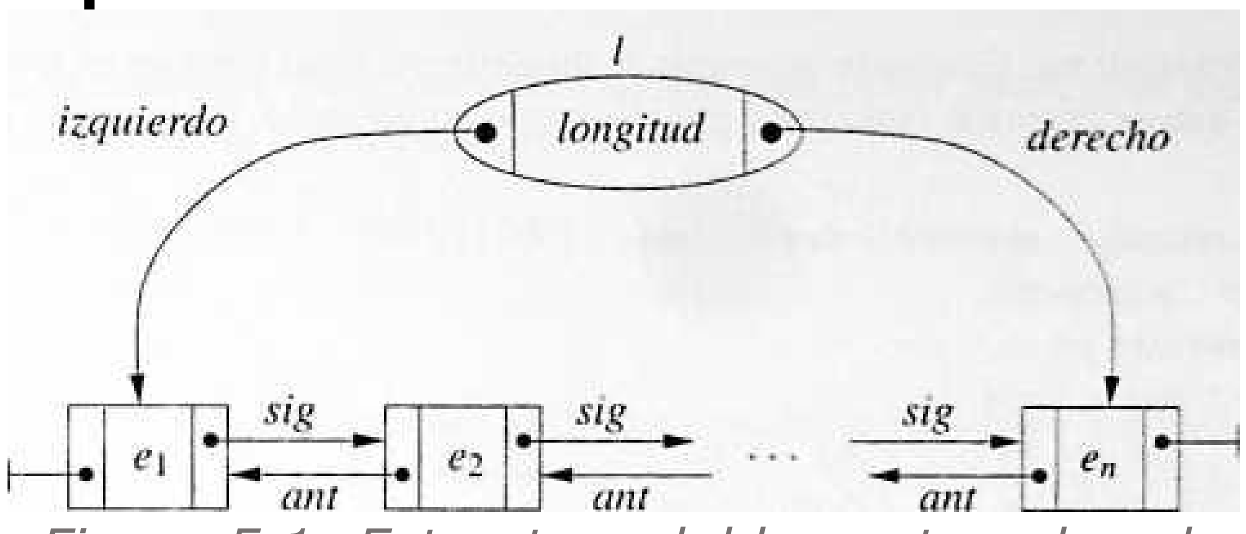


Figura 5.1: Estructura doblemente enlazada para listas.

Nótese que esta estructura doblemente enlazada es similar a la estructura dinámica utilizada para implementar colas dobles en el Apartado (b) del Ejercicio 4.8. excepto que se ha añadido un campo adicional *longitud* para poder calcular esta información en tiempo constante.

La lista vacía no tiene elementos, por lo que su longitud es 0. y como no hay

elementos, los punteros *izquierdo* y *derecho* son nulos.

fun lista-vacía() **dev** *I : lista* { 0(1))
  i.longitud := 0
  I.izquierdo := nil ; *i.derecho* := nil
ffun

Añadir un elemento por la izquierda consiste en reservar memoria para un nuevo nodo, almacenar en él el nuevo elemento, y enlazarlo con la estructura, distinguiendo el caso en el que la lista inicial sea vacía, en cuyo caso el elemento añadido también es el derecho. En cualquier caso la longitud de la lista aumenta en una unidad. Añadir un elemento por la derecha es la operación ''simétrica".

proc añadir-izq(e *e : elemento, I : lista)*
(0(1) } **var** *p : enlace-lista*
  reservar(p)
  p j *.valor := e ; p* t *.ant* := nil
  si / *izquierdo* = nil **entonces** { / es vacía }
    p f *.sig* := nil ; *i derecho := p*
  si no
    p f *.sig := /.izquierdo: (I .izquierdo)* t *.ant := p*
  fsi
  i .izquierdo : = p
  i.longitud := I.longitud + **I**
fproc

proc añadir-der(/ : *lista,* **e** *e : elemento)*
(0(1)) **var** *p : enlace-lista*
  reservar(p)
  pf.valor := e ; p t *.sig* := nil
  si /*.derecho* = nil **entonces** *p* t *.ant* :=

nil ; *i.izquierdo p*

si no *p* f *.ant := i.derecho ; (i.derecho)*
f *sig := p*

fsi

i.derecho := p

i.longitud := i.longitud + 1

fproc

Construir una lista unitaria a partir de un elemento dado consiste en crear una nueva lista con un único nodo que es apuntado tanto por *izquierdo* como por *derecho.* La longitud es I.

fun unitaria(e : *elemento)* dev *I : lista* { 0(1) } **var** *p : enlace-lista*
reservar(p)
p t *.valor := e*
p'f.ant := nil; *p^.sig* := nil
i.izquierdo := p ; i.derecho := p
i.longitud := l
ffun

El elemento más a la izquierda de una lista no vacía se obtiene accediendo al campo *valor* del nodo apuntado por *izquierdo.* Si la lista es vacía se produce un mensaje de error. Para eliminar el izquierdo se libera el primer nodo de la estructura, dejándola bien enlazada. Las operaciones derecho y elim-der son las “simétricas”.

fun izquierdo(/ : *lista)* dev *e : elemento* (® (l))
si *i.izquierdo* = nil **entonces**
error(Lista vacía)
si no *e := (/ .izquierdo)* f *.valor*
fsi
ffun
proc elim-izq(/ : *lista)* (0(1)) **var** *p : enlace-lista*
si *i.izquierdo* = nil **entonces**
error(Lista vacía)
si no
p := i.izquierdo; i.izquierdo p^.sig

si *i.izquierdo* = nil **entonces**
i.derecho nil
si no *(i.izquierdo)* f *.ant ;*= nil
fsi
i.longitud := i.longitud — 1
liberar(p)
fsi
fproc
fun derecho(í *: lista)* **dev** *e : elemento* (0(1))
si *i.derecho* = nil **entonces**
error(Lista vacía)
■**si no** *e := (i.derecho)* f *.valor*
fsi
ffun
proc elim-der(/ : *lista)* (0(1))
var *p : enlace-lista*
si *i.derecho* = nil **entonces**
error(Lista vacía)
si no
p i.derecho; i.derecho ;= p^.ant
si *i.derecho* = nil **entonces**
i.izquierdo := nil
si no *(i.derecho)* f *.sig ;*= nil
fsi
i.longitud := i.longitud — 1

fsi
fproc

liberar(p)

Una lista es vacía cuando los punteros (uno o ambos, pues ambas condiciones son equivalentes) son nulos pues eso significa que no hay nodos en la estructura.

fun es-lista-vacía?(/ : *lista)* dev *r : bool*
(0(1) |
  r := (l-izquierdo = nil)
ffun

La operación concatenar se implementa como una función que devuelve una lista, resultado de concatenar las dos listas que recibe como argumentos. La lista resultado comienza siendo una copia de la primera lista argumento (utilizando la función copiar-lista del Ejercicio 5.4). A continuación se recorre la sucesión de nodos de la segunda lista, añadiendo cada elemento por la derecha a la lista resultado. El coste en tiempo de concatenar es lineal respecto a la suma de las longitudes de las listas concatenadas, ya que copiar tiene un coste lineal respecto a la longitud de la lista copiada. Obsérvese que para abreviar el código de tratamiento de punteros para añadir un elemento se utiliza una llamada a añadir-der.

fun concatenaría-. y : *lista)* **dev** *z : lista*
(0(longitud (.r) + longitud(y))) **var** *p enlace-lista*
  - := lista-vacíaí)
  p := x .izquierdo
  mientras *p* nil **hacer**

añadir-der(.t, *p* f *.valor)*
p ■— *p^.sig*
fmientras
p y *izquierdo*
mientras *p* nil **hacer**
añadir-der(-, *p^ valor)*
p := pf *sig*
fmientras ffun

Si se implementara esta operación mediante un procedimiento, se podría conseguir un coste constante modificando el primer argumento para que también fuera el resultado, para lo cual bastaría encadenar el nodo *derecho* de ,v con el *izquierdo* de y y sumar las longitudes.

La longitud ele una lista se obtiene consultando el campo correspondiente, con un coste en tiempo constante.

fun longitud!/ : *lista)* **dev** *n : nat* (0(1))
n := /.longitud
ffun

Utilizando la representación dinámica del Ejercicio 5.3. implementar una función que reciba una lista ■ y devuelva una copia de esta en posiciones de memoria diferentes, y un procedimiento que libere la memoria ocupada por una lista, dejando a esta vacía.

-------------------Solución------------------

La función copiar-lista y el procedimiento anular-lista son prácticamente idénticos a los dados en el Ejercicio 4.8 para el caso de las colas dobles. Solo cambian los nombres de los

tipos de las variables utilizadas y el hecho de que en este caso también hay que copiar el campo *longitud.*

Extender la implementación del Ejercicio 5.3 implementando (con acceso a la representación) las operaciones especificadas en el Ejercicio 5.2.

---------------------Solución---------------

Para averiguar si un elemento está en una lista se recorre la estructura de nodos enlazados mediante una variable *p* de tipo *enlace-lista,* hasta que se encuentre el elemento o se acabe la estructura *(p* = nil). En cada nodo del recorrido se consulta si el elemento en el campo *valor* es igual al elemento *e* dado.

fun está?(e : *elemento. I : lista)* **dev** *b : bool* (O (longitud(Z)))
var *p : enlace-lista*
b := falso ; *p := i.izquierdo*
mientras *p £* nil A -■i> **hacer**
b (e = p f *.valor)*
p := p'f.sig
fmientras
ffun

La función posición para localizar un elemento es muy similar. Al recorrer la lista vamos guardando en una variable auxiliar *i* la posición del elemento en el que nos encontramos. Si el valor apuntado es igual al que buscamos, asignamos el valor de *i* al parámetro de salida *n.* Obsérvese cómo se utiliza el valor de *n,* inicializado a 0, para controlar en la

condición del bucle si el elemento ya se ha encontrado. Además, si el elemento no está en la lista, se devuelve dicho valor nulo.

fun posición(<? : *elemento, I : lista)*
dev *n : nat* (0(longitud(/))) **var** *p : enlace-lista*
n := 0 ; *i* := 0 ; *p* *i.izquierdo*
mientras *p* yé nil A *n* = 0 **hacer**
si *e* = *p* f *.valor* **entonces** *n* := *i* **fsi**
P '■= *P* t *sig* I *i* := *i* + 1
fmientras
ffun

Para contar el número de apariciones de un elemento hay que recorrer toda la lista.

fun repeticiones (e : *elemento,! : lista)* **dev** *n : nat* (0(longitud(Z)))
var *p : enlace-lista*
n := 0; *p := i.izquierdo*
mientras *p* nil **hacer**
si *e = p^.valor* **entonces** *n := n +* 1 **fsi**
p := p f *.sig*
fmientras
ffun

Para eliminar todas las apariciones de un elemento se recorre la estructura de nodos enlazados. Cuando se encuentra un nodo con un elemento igual al que hay que eliminar, hay que modificar todos los enlaces para evitar dicho nodo que se libera, tratando apropiadamente los casos en que este es el primero o el último. La longitud va decreciendo a

medida que se eliminan nodos.

proc eliminarte *e : elemento. I : lista)* { 0(longitud(/)) }

var *p. q : enlace-lista*

p := í f *.izquierdo*

mientras *p* / nil **hacer**

si *p* f *.valor = e* **entonces**

si p = /1 *.izquierdo* **entonces**

/1 *.izquierdo := p* -f *.sig ; p* f *.sig*

f *.ant* := nil

si no *p* f *.ant* f *.sig* := p f *.sig*

fsi

si *p = 1*1 *.derecho* **entonces**
l f *.derecho := p* f *.ant ; p* f *.ant* j
.sig := nil
sino *p* T *.sig* f *.ant := p\.ant*
fsi
q := P : P ■= *p'f.sig:* liberarte/)
I.longitud := i.longitud — 1
si no *p := p* **f** *.sig*
fsi
fmientras
fproc

Para determinar si dos listas son iguales, primero se comprueba si tienen la misma longitud. Si es así. se utilizan dos punteros auxiliares *p* y *q* para recorrer completamente cada una de las listas, comprobando si los elementos apuntados por ambos son iguales.

fun igual?(.r. y : *lista)* **dev** *b : bool* (©(longitud(.r)) }
var *p. q : enlace-lista*
si *x.longitud y.longitud* **entonces**
b = falso
si no
b := cierto
p := **v** *izquierdo : q := y.izquierdo*
mientras *p* / nil A *b* **hacer**
b := (p f *valor = q* f *.valor)*
p := p **t** *sig ; q := q* **f** *.sig*
fmientras
fsi
ffun

Para comprobar si una lista es capicúa utilizamos dos enlaces: *p* recorrerá los elementos de la lista de

izquierda a derecha, y *q* recorrerá los elementos de derecha a izquierda. Ya que no se puede comprobar fácilmente cuándo se cruzan los punteros en su recorrido, utilizamos una variable *ni* para contar cuántos elementos hay desde *p* hasta *q* (ambos incluidos). En tanto *ni* > 1. los elementos apuntados por *p y q* tienen que ser iguales para que la lista sea capicúa. Nótese que el bucle para tan pronto como se encuentran dos elementos distintos.

```
fun es-capicua?(/ : lista) dev b : bool {
0(longitudf/)) )
  var p. q : enlace-lista
    b := cierto
    p := i.izquierdo : q := i.derecho
    ni := l.longitud
    mientras ni > 1 A b hacer
      b := (p f valor = q f .valor)
      p := p f .sig ; q := q j" .ant
      m := m — 2
    fmientras
  ffun
```

Este algoritmo es similar al visto en el Ejercicio 4.9 (Apartado (c)) para comprobar si una frase es palíndroma, aunque allí se utilizaba una cola doble a la que se accedía de forma abstracta, sin el manejo de los punteros de la representación.

' (a) Especificar las siguientes operaciones sobre listas:

- calcular la lista formada por los *n* elementos más a la izquierda, y
- calcular la lista formada quitando los n elementos más a la izquierda.

(b) Implementar dichas operaciones como procedimientos utilizando de forma abstracta las operaciones especificadas en el Ejercicio 5.1.

(c) Implementarlas utilizando directamente la implementación dinámica de las listas del Ejercicio 5.3.

-----------------Solución------------------

Apartado (a)

Para poder “coger" o “tirar" cierto número de elementos del comienzo de una lista no se requiere nada especial de los elementos almacenados en ella, por lo que utilizaremos el parámetro *ELEM(*Sección 1.1.5). Especificamos las dos operaciones como totales: cuando la lista tiene menos elementos de los que se quieren coger, se devuelve toda la lista; y si la lista tiene menos elementos de los que se quieren tirar, se devuelve la lista vacía.

especificación *LISTAS-EXTRAER[ELEMi*

usa *LISTASfELEM], NATURALES*

operaciones

coger : *nat lista —> lista*

tirar : nat *lista —> lista*

variables

e : elemento

I : lista
n : nat

La especificación de las operaciones distingue casos según que la lista sea vacía o no (vamos a utilizar como constructoras la lista vacía y añadir por la izquierda). Si la lista no es vacía se distinguen casos según el natural que indica cuántos elementos hay que tratar sea cero o positivo.

ecuaciones
coger(n, []) = []
coger(0. *e:l) = []*
coger(n. <?:/) = *e* : coger(/t — 1. /)
<= *n* > 0
tirar (ti. []) = []
tirar(O. *e:l) = e:l*
tirar(n. *e:l)* = tirar(n — 1,/) <= *n* >
0
fespecificación

Apartado (b)----------------

Ambas operaciones se implementan como procedimientos que modifican la lista argumento. La operación coger es muy simple: hay que descartar sucesivamente el último elemento hasta que la lista tenga *n* elementos.

proc cogerte *n : nat, I : lista)* {
0(longitud(/) - n))
mientras longitud (/) > *n* **entonces**
elim-der(í)
fmientras
fproc

Para implementar tirar hay que eliminar *n* veces (o menos, si la lista es

más corta) el elemento más a la izquierda.
proc tirarle *n : nat. I : lista)* (0(n)) *ni := n* **mientras** *ni* > 0 A -■es-lista-vacía?!/) **hacer** elim-izq(/) ; *ni := ni* — I **fmientras**
fproc

El coste de coger está en © (longitud!/) — *n)* porque se recorre la lista desde la derecha eliminando elementos hasta que solo quedan *n.* El coste de tirar está en *&(n)* porque se eliminan *n* elementos empezando por la izquierda.

Apartado (c)-----------------------

Como ahora tenemos acceso a la representación dinámica de la lista, consideramos algoritmos que indican directamente lo que se ha de hacer con los diversos enlaces. En la implementación de coger se recorren de derecha a izquierda los elementos que hay que eliminar de la lista. Para tirar se van eliminando elementos por la izquierda, hasta que se hayan eliminado tantos como se quieran descartar.

proc cogerle *n ; nat. I : lista)* (©(longitud!/) — *n) }* **var** *p. q : enlace-lista p := / .izquierdo*
mientras *i .longitud* > *n* **entonces**
q ⁝= *P'*■ *P — P* T -
uu' '• liberarle/)
l.longitud := l
.longitud — l
fmientras *l.derecho*

:= p
fproc
proc tirarle *n : nat. I : lista)* **(0(n))**
var *p.q* ■ *enlace-lista*
si *n > /.longitud* **entonces** anular(Z)
si no
p i.izquierdo
para *i* = I **hasta** *n* **hacer**
q := p \ p := p f *.sig ;*
liberarle/) : */.longitud :=*
l.longitud — \ **fmientras**
/.izquierdo := p : (/.izquierdo) f *.ant*
:= nil **fsi**
fproc

El coste de coger está en ©(longitud!/) - *n)* ya que se recorren (y se eliminan) todos los nodos de la lista menos los *n* que permanecen. El coste de tirar está en ©(n) porque se recorren los *n* elementos que se tiran. Así pues, los costes son los mismos que en el apartado anterior.

5.7. Suponiendo que los elementos de las listas tienen una relación de orden, especificar una operación que compruebe si una lista está ordenada y operaciones que ordenen de menor a mayor los elementos de una lista según:

- el método de ordenación por inserción,
- el método de ordenación por mezclas *(mergesort).* y
- el método de ordenación rápida *(quicksort).*

-------------------Solución-------------------

Para determinar si una lista está ordenada, o para ordenar una lista, necesitamos que los elementos que forman la lista tengan una relación de orden total (<). Especificamos el siguiente parámetro, que define los tipos con operaciones de orden < y >, y las propiedades que estas relaciones tienen que satisfacer.

parámetro *ELEM<*
usa *BOOLEANOS*
tipos *elemento*
operaciones
_ < _ : elemento elemento —> bool
_> _ : elemento elemento —> bool
variables
e, f. g : elemento
ecuaciones
e< e —cierto (reflexividad }
e< f =cierto 4=e<gAg</ { transitividad)
e> *f* = ~’(e < *f)*
e> f =falso 4= *f* > *e* { antisimetría)
e< f =cierto <= -•(/ < e) (totalidad }
fparámetro

Obsérvese que la última ecuación es equivalente a esta otra

e < *f v f* <*e* = cierto

El TAD de las listas que se pueden ordenar se especifica con un parámetro *ELEM<.* Para especificar las

operaciones de ordenación se utiliza una serie de operaciones privadas, cuyo significado se explicará a continuación. Aunque en principio la operación insertar-ord (que inserta un elemento en una lista ordenada) podría declararse también como privada en esta especificación, no se declara así porque la operación tiene sentido en sí misma, y se utilizará en otros ejercicios más adelante.

especificación *LISTAS-ORDENADAS[ELEM<]*
usa *BOOLEANOS. LISTAS[ELEM<]*
operaciones
ordenada? *: lista —> bool*
ordenar-inserción : *lista —>* • *lista*
ordenar-mezcla *: lista —> lista*
ordenar-rápida *: lista —» lista*
insertar-ord *: elemento lista —> lista*
operaciones privadas
mezclar-ord *: lista lista —> lista*
elem-menor-ig : *elemento lista —> lista*
elem-mayor : *elemento lista —» lista*
variables
e, f : elemento
x, y. lista
ecuaciones
ordenada?([]) = cierto
ordenada?(e: [ ]) = cierto
ordenada?(e:/:a) = *e<f* A

ordenada?)/: .v)

Ordenar por inserción consiste en, partiendo de la lista vacía, insertar manteniendo el orden los ele mentos sucesivamente. Las siguientes ecuaciones representan esta idea de forma recursiva, usando It operación insertar-ord. Esta operación *no es parcial,* pero solamente tiene el efecto deseado de insertar e

primer argumento en la posición adecuada de la lista dada como segundo argumento cuando esta es una lista ordenada, lo que está garantizado cuando se usa en la definición de ordenar-inserción. Para especificar esta y las siguientes operaciones utilizaremos como constructoras de las listas la lista vacía y añadir por la izquierda.

ordenar-inserción ([|) = []

ordenar-inserción*(e :x)* = insertar-ord(e, ordenar-inserción(.r))

insertar-ord(e, [ ]) = [<?]

insertar-ord(e, *f* : x) = *e: (f* ; x) 4= *e* < *f*

insertar-ord(e, *f* : x) = *f :* insertar-ord(e, x) <=<?>/

La ordenación por mezclas consiste en dividir por la mitad la lista, ordenar cada mitad, y mezclar de forma ordenada las dos sublistas ordenadas. Para dividir la lista se utilizan las operaciones coger y tirar definidas en el Ejercicio 5.6. La función mezclar-ord, sin ser parcial, solamente produce

el resultado deseado cuando sus argumentos son listas ordenadas.

ordenar-mezcla([ ]) = |]

ordenar-mezcla(e : [ ]) = *e : [ ]*

ordenar-mezcla(x) = mezclar-ord(ordenar-mezcla(coger(longitud(.r) div 2. .v)).

ordenar-mezcla(tirar(longitud(x) div 2. x)))

<= longitud(x) > 2

mezclar-ord(x, [ ]) = x

mezclar-ordíl]. y) = y

mezclar-ordfe x. /:v) = *e* : mezclar-ord(x, *f*: y) 4= *e<f*

mezclar-ord(e : x. *f :* y) = /:

mezclar-ord(e :x, y) 4= *e* > *f*

La ordenación rápida consiste en elegir un elemento de la lista como *pivote* (en nuestro caso, el primero de la lista), dividir el resto de la lista en dos listas (una con los elementos menores o iguales que el pivote y otra con los elementos mayores que el pivote), ordenar las dos sublistas y concatenarlas, colocando entre ellas el pivote.

ordenar-rápida(( ]) = []

ordenar-rápida(e: v) — ordenar-rápida(elem-menor-ig(e. x))
-H- je] -H- ordenar-rápida(elem-mayor(e. x))

elem-menor-ig(e. [ ]) = []

elem-menor-ig(e. *f* : x) = *f* : elem-menor-ig(e. x) 4= *f* < *e*

elem-menor-ig(e./:x) = elem-menor-ig(e.x) 4= *f* > *e*

elem-mayor(e, |]) = []

elem-mayor(e. *f* :x) = elem-mayor(e. x) 4= *f*<*e*

elem-mayor(c. *f* x) = *f* : elem-mayor(e. x) <= *f*>*e*

fespecificación

Queremos extender el TAD de las listas de forma que podamos acceder a cualquiera de los elementos de la lista a partir de su posición, bien para consultarlo, modificarlo, insertarlo o eliminarlo. Especificar las operaciones descritas a continuación:

- consultar el elemento en una posición,
- insertar un elemento en una posición,
- eliminar el elemento en una posición, y
- modificar el elemento en una posición.

----------------Solución-----------------

Estas listas con acceso por posición pueden estar formadas por elementos de cualquier tipo, por lo que utilizaremos el parámetro *ELEM.* Las operaciones se definen como parciales: solo están definidas si la posición solicitada tiene sentido para la lista en cuestión. Suponiendo que los elementos empiezan a numerarse desde el 1. las posiciones válidas estarán entre 1 y la longitud de la lista dada, salvo en el caso de insertar, donde también se permite que la posición

sea uno más que la longitud de la lista, lo que permitirá añadir al final de la lista.

especificación *LISTAS-POSICIÓN[ELEM]*

usa *NATURALES, LISTAS[ELEM]*

operaciones

_[_] : lista nat —> ,, elemento

insertar : *lista nat elemento* ■—> *p* *lista*

eliminar : *lista nat —> p lista*

modificar : *lista nat elemento —> p lista*

variables

e. f : elemento

I : lista

i : nat

Las operaciones de consulta (_[_]), inserción y eliminación se definen en el caso no erróneo distinguiendo casos según la posición sea 1 o mayor. Utilizamos como constructoras de las listas la lista vacía y añadir por la izquierda.

ecuaciones

l[i] = error <= *i* == 0 v *i* > longitud!/)

<e:/)[I] = *e*

(?:/)[/] = /[/ — 1] <= 1 < *i A i* < longitud!?:/)

insertar!/. i. *e)* = error <= *i* == 0 V *i* > longitud!/) + 1

insertar!/. 1, e) = *e.l*

insertar!?:/./ ./) — ?: insertar!/, í - 1,/) <= 1 < *i* A í < longitud!? :/) + 1

eliminar!/, í) = error <= *i* == 0 v *i* > longitud!/)

eliminar!?:/, 1) = /

eliminar!?:/, i) = ?:eliminar!/, i — 1) <= 1 < i A *i* < longitud!? :/)

La operación modificar se especifica utilizando directamente eliminar e insertar.

modificar!/, *i, f)* = insertar(eliminar(/, /). *i. f)*

fespecificación

(a) Especificar una operación que transforme una lista de naturales en una lista de booleanos. de forma que los números pares se transformen en cierto y los impares en falso.

(b) Especificar una operación que eleve al cuadrado todos los elementos de una lista de enteros.

(c) Especificar una operación que, dada una lista de listas, devuelva una lista de naturales de forma que cada lista se convierta en su longitud.

(d) Generalizar los tres apartados anteriores considerando una operación genérica de listas en listas definida en una especificación parametrizada por un parámetro adecuado.

-----------------Solución------------------

En todos los apartados siguientes vamos a utilizar como constructoras de las listas la lista vacía y añadir por la izquierda.

Apartado (a)-----------------

La operación par-impar? se define en una especificación que utiliza las listas de naturales y las listas de booleanos, ya que la operación recibe como argumento

una lista de naturales y la transforma en una lista de booleanos. Las ecuaciones distinguen casos según las constructoras. Si la lista (de naturales) es vacía se devuelve también la lista vacía (de booleanos) que se denota de la misma forma. Si no lo es. se aplica la operación de los naturales es-par? (véase el Ejercicio 1.2) al elemento más a la izquierda, y se sigue de forma recursiva con el resto de la lista.

especificación *PAR-IMPAR*
usa *LISTAS[NATURALESJ. LISTAS[BOOLEANOS]*
operaciones
  par-impar? . *hsta[nat] —> lista[booli*
variables
  e : nat
  l : lista[nat]
ecuaciones
  par-impar?([ ])   = []
  par-impar?(*e :/)* = es-par?(e): par-impar?(/)
fespecificación

Apartado (b)-------------------------

En este caso la operación cuadrados convierte una lista de enteros en otra lista de enteros, por lo que se define en una especificación que utiliza estas listas. Las ecuaciones siguen el mismo patrón que en el caso anterior. Si la lista argumento es vacía se devuelve la lista vacía. Si no lo es. se construye una lista calculando el cuadrado del elemento más a la izquierda, y siguiendo de forma recursiva con el resto de la lista.

especificación *CUADRADOS*
usa *LISTAS[ENTEROS]*
operaciones
cuadrados *listafent] —» lista[ent]*
variables
e : ent
I : lista[ent]
ecuaciones
cuadrados![I) = I)
cuadrados(e :/) = *(e * e)*: cuadrados(í)
fespecificación

Apartado (c)--------------------------

En este caso el argumento es una lista de listas de valores de cualquier tipo. Por tanto, la especificación es parametrizada sobre el parámetro *ELEM.* Se utilizan las listas de listas de elementos (tipo *elemento* de *ELEM)* y las listas de naturales. Las ecuaciones siguen el mismo patrón que en los casos anteriores: aplicar la función de listas longitud (véase el Ejercicio 5.1) al elemento más a la izquierda y continuar con el resto de forma recursiva.

especificación *LONGITUDES[ELEM]*
usa *LISTAS[LISTAS[ELEM]]. LISTAS[NATURALES]*
operaciones
longitudes : *lista[lista] —> lista[nati*
variables
e : lista
I : lista[lista]
ecuaciones
longitudes!!]) = []
longitudes!?:/) = longitud(e): longitudes!/)

fespecificación

Apartado (d)-----------------------------

Lo que tienen en común las operaciones especificadas en los tres apartados anteriores es que convierten una lista de elementos de un tipo *A* en una lista de elementos de un tipo B (B coincide con *A* en el Apartado (b)). Además todas ellas devuelven la lista vacía cuando el argumento es la lista vacía, y cuando la lista no es vacía, aplican una función f a su primer elemento, y operan con el resto de forma recursiva. La idea intuitiva es que estas operaciones convierten una lista [o i *.ai. ..* . «„] en la lista [f(«l). fien) ......f(a_n)] para una determinada función f, operación genérica que en programación funcio
nal recibe el nombre de *map.*

Definimos un parámetro *FUNCIÓN* que declara dos tipos *Ay B y* una operación f de *A* en B. No es necesario requerir nada especial sobre ninguno de estos elementos.

parámetro *FUNCIÓN*
tipos A, B
operaciones
  f : *A* —> B
fparámetro

A continuación especificamos la operación genérica map que generaliza las operaciones de los apartados anteriores, en una especificación parametnzada con el parámetro *FUNCIÓN.* Las ecuaciones siguen el mismo patrón utilizado en los tres casos

concretos anteriores.

especificación *MAP[FUNCIÓN]*
usa *LISTAS[A], LISTAS[B]*
operaciones
map : *lista[A] —» lista[B]*
variables
e : A
I : lista[A]
ecuaciones
map(ll) = I]
map(e:/) = f(e):map(/)
Fespecificación

Especificar una operación que dada una lista de naturales devuelva la suma de todos los elementos.

(a) Especificar una operación que dada una lista *L* de listas devuelva una lista formada por la concatenación de todas las listas de *L.*

(b) Especificar una operación que reciba una lista cuyos elementos representen los dígitos en base *b* de un número *n* (con el dígito menos significativo a la izquierda) y devuelva la representación de n como número natural.

(c) Generalizar los tres apartados anteriores para obtener una operación genérica sobre listas, definida en una especificación parametrizada por un parámetro adecuado.

-----------------Solución-----------------

En los siguientes apartados utilizamos como constnictoras de las listas la lista vacía y añadir por la izquierda.

Apartado (a)-------------

La operación suma-todos convierte una lista de naturales en un natural. Si la lista es vacía, la suma de sus elementos es 0. Si la lista no es vacía, la suma de sus elementos se obtiene sumando el elemento más a la izquierda al resultado de sumar los elementos del resto de la lista.

especificación *SUMA-TODOS*
usa *LISTAS[NATURALESJ. NATURALES*
operaciones

suma-todos *hsta[nat] —> nat* **variables**
n : nat
I : lista[nat]
ecuaciones
suma-todos((|)   =0
suma-todos(/i :/) = *n* + suma-todos(í)
fespecificación

Apartado (b)

Llamaremos aplanar a la operación que concatena todas las listas elementos de una lista de listas *L.* El resultado de aplanar una lista vacía es la lista vacía (de elementos). Si la lista *L* no es vacía, se puede aplanar concatenando su elemento más a la izquierda (una lista) al resultado de aplanar el resto de la lista /.. Las listas básicas pueden ser de elementos de cualquier tipo, por lo que escribiremos una especificación parametrizada. con parámetro *ELEM* (Sección 1.1.5).

especificación *APLANAR[ELEM]*
usa *LISTAS[USTAS[ELEMJ], LISTAS[ELEM]*
operaciones
aplanar *lista[listal —> lista* **variables**
e : lista
I : lista[lista]
ecuaciones
aplanar),])= [ ]
aplanarte:/) = e-H-aplanar)/)
fespecificación

Apartado (c)------------------------

Evaluar la representación en base *b* de un número (dada como una lista) para obtener su valor numérico consiste en

evaluar un polinomio. Si la lista es [ZQ, Z|..... *In*~\, /„]. el valor buscado es J2"=o Z,- * . Podemos utilizar el método de Horner para evaluar polinomios (véase el Ejercicio 2.4), sumando el coeficiente del término de menor grado (dígito menos significativo, el de más a la izquierda de la lista) al valor obtenido multiplicando por *b* el resultado de evaluar el resto del polinomio.

especificación *EVALUAR*
usa *LISTAS[DÍGITOS], NATURALES*
operaciones
evaluar : *lista[dígito] —> nat*
variables
d : dígito
l : Hsta[digito]
ecuaciones
evaluar([]) = 0
evaluar(<Z : Z) = *d* + *b** evaluar(Z)
fespecificación

Apartado (d)----------------------

Los apartados anteriores tienen en común que las tres operaciones especificadas reciben una lista de valores de un tipo *A* y devuelven un único valor de un tipo *B.* en general, distinto de *A-* Además, cuando la lista argumento es vacía, se devuelve un valor distinguido de tipo B. mientras que cuando la lista no es vacía, el resultado se obtiene combinando con una función f el primer elemento con el resultado de haber combinado el resto de la lista. Dada una lista j«j, as a„], el resultado de la combinación es

«i f *(as* f (... f (a„ f v)...) (donde

hemos utilizado f como un operador infijo y v es el valor distinguido mencionado antes), operación que en programación funcional recibe el nombre de *fold.*

Especificamos un parámetro con estos elementos.

parámetro *FUNCIÓN-BIN* **tipos** *A, B*
operaciones
v : —>8
_f_ : *AB —> B*
fparámetro

Ahora especificamos la operación fold que combina los elementos de una lista argumento por medio de la operación f recibida dentro del parámetro de la especificación.

especificación *FOLD[FUNCIÓN-BIN]*
usa *LISTAS[A]*
operaciones
fold : *lista[A]* —> B
variables
a : *A*
I : listafA]
ecuaciones
fold([j) = v
fold(a: Z) = affold(Z)
'fespecificación

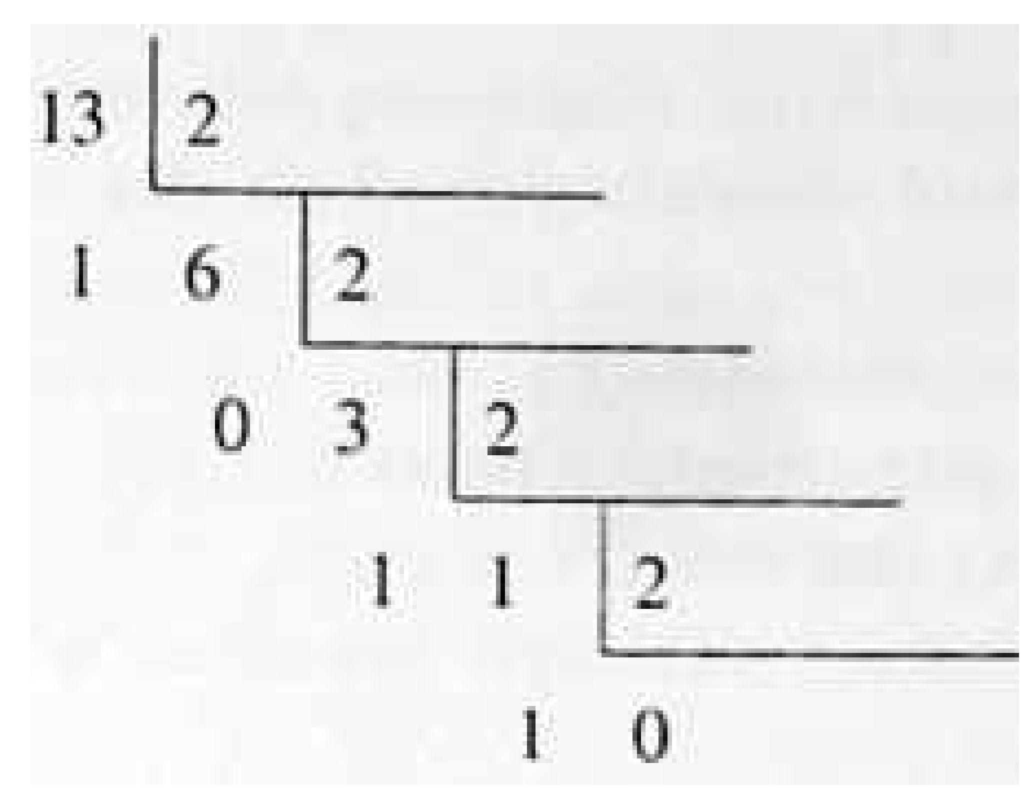


Figura 5.2: Ejemplo de cambio de base.

(a) Diseñar una función que, dados un número natural *n* y una base *b* ($2 < b < 9$), devuelva una lista cuyos elementos sean los dígitos de la representación de *n* en base *b,* con el dígito más significativo a la izquierda.

(b) Diseñar una función que realice la operación inversa, es decir, que dada una lista que represente un número escrito en base *b.* devuelva su valor como número natural.

--------------------Solución---------------

Apartado (a)

La idea para obtener la representación en base *b* de un número *n* consiste en ir dividiendo sucesivamente *n* por *b* mientras el cociente sea distinto de 0. Los restos de estas divisiones forman los dígitos de la representación en base *b.* obteniéndose primero el menos significativo. Como ejemplo, la división de la Figura 5.2 muestra cómo se obtiene la representación en base 2 del

número 13: 1101.

El siguiente algoritmo implementa esta idea. Como los dígitos se van obteniendo en orden inverso al que nos piden, se van añadiendo a la lista resultado por la izquierda. Así el más significativo quedará a la izquierda, como pide el enunciado.

fun cambiar-a-base(/i : *nat. b* : 2..9)
dev *I : lista[dígito]* (<9(log/,n) }
/ := lista-vacía()
si *n* = 0 **entonces** añadir-izq(0. /)
si no *ni := n* **mientras** *ni* > 0 **hacer**
añadir-izq(m mod *b. /) ni := ni* div *b*
fmientras
fsi
ffun

El coste está en (-) (log¿ *n) ya* que ese es aproximadamente el número de veces que se puede div idir *n* por *b,* que coincide con el número de elementos en la lista resultado.

Apartado (b)-----------------------

Evaluar la representación en base *b* de un número (dada como una lista) consiste en evaluar un polinomio. como hemos indicado en la solución del Apartado (c) del Ejercicio 5.10. para lo que podemos utilizar el método de Horner (véase el Ejercicio 2.4). Como la lista se va destruyendo al recorrerla con izquierdo, el algoritmo copia inicialmente la lista dada en una variable auxiliar;.

(l no es vacía)

fun evaluar-en-base(/ : *lista[dígito], b :* 2..9) **dev** *n : nat* (<9 (longitud(Z)))

var *z : lista*
n := 0 : z := copiar-lista(/)
mientras -■es-lista-vacía?(z) hacer
*n := n * b* + izquierdo(z)
elim-izq(z)
fmientras
fftin

El coste es lineal respecto a la longitud de la lista argumento que se recorre completamente.

En la especificación del Ejercicio 5.1, el tipo de listas solo admite listas de elementos sobre un mismo tipo parámetro, como por ejemplo [1,2, 3] en el caso de listas de naturales o | [4], [5], [6|| en el caso de listas de listas de naturales, pero *no* es posible generar listas de la forma [7. [8[. 9|, cuyas componentes son o bien elementos básicos o bien listas del mismo estilo. Especificar el TAD de *listas generales* sobre un tipo parámetro dado, donde una lista general o bien es vacía, o bien está formada por elementos del tipo parámetro o listas generales. Las tres listas [1,2, 3], [[4], [5], [6]] y [7, [8], 9] son ejemplos de listas generales sobre los naturales, así como la lista *L* = [[1, 2, 3], [[4], [5], [6]], [7. [8], 9). []].

Determinar operaciones constructoras adecuadas para este TAD, y especificar las siguientes operaciones:

- devolver la componente más a la izquierda de una lista general.
- eliminar la componente más a la

izquierda,

- devolver el elemento básico (o sea, perteneciente al tipo parámetro) más a la izquierda dentro de una lista general, por ejemplo 1 para la lista *L,*
- devolver el elemento básico más a la derecha, por ejemplo 9 para la lista *L.*
- concatenar dos listas generales.
- calcular la longitud de una lista general, por ejemplo 4 para la lista *L.*
- invertir una lista general solo al nivel más alto, por ejemplo para la lista *L* anterior se obtendría [[1,[7,[8].9],[[4],[5],[6]],[1,2.3]],
- invertir una lista general, invirtiendo asimismo cada una de sus componentes, por ejemplo para la lista *L* se obtendría [[]. [9, [8], 7], [[6], [5], [4j[, [3, 2. 1]].
- calcular el nivel de atildamiento de una lista general, por ejemplo para la lista *L* es 3. y
- calcular la longitud máxima de todas las listas que forman parte (tal vez anidadas) de una lista general, por ejemplo 4 para la lista *L* porque la lista total es la más larga.

------------------Solución------------------

En todas las estructuras lineales que se han especificado hasta ahora (conjuntos, pilas, colas, listas, etc.) ha sido posible utilizar como constructoras una operación para crear la estructura vacía junto con otra operación para ir añadiendo elementos a una estructura, de forma que cualquier estructura del

tipo dado se obtiene partiendo de la estructura vacía añadiendo elementos sucesivamente. En el tipo de datos de listas generales que se pide especificar, la idea es la misma, pero la diferencia es que todas las componentes de la estructura no son de un mismo tipo pues además de añadir elementos básicos (del tipo *elemento* dado como parámetro) también hay que contar con la posibilidad de añadir una lista general a otra lista general, de forma que la primera lista se convierte en la componente más a la izquierda de la segunda (es importante no confundir esta operación con la concatenación de dos listas, en la cual todos los elementos se quedan al mismo nivel y no una lista anidada dentro de otra).

Podría pensarse en tener una única operación para añadir, cuyo segundo argumento es siempre una lista general y cuyo primer argumento es o bien un elemento básico o bien una lista general: sin embargo, esto no es posible en el lenguaje de especificación que estamos utilizando, porque no tenemos forma de definir un tipo como unión de otros dos tipos. Por esta razón tenemos que considerar dos operaciones diferentes para añadir: una. que denotamos como en las listas normales, para añadir un elemento básico a una lista general, y otra, que denotamos _ :: para añadir una lista general a otra lista general. Junto con la operación para crear la lista general vacía, que denotamos [ ]. tenemos en total tres constructoras *libres.*

Según la lista general que se tenga, la componente más a la izquierda puede ser o bien un elemento básico, o bien una lista general. De nuevo, por no tener un tipo unión, no podemos tener una única operación (parcial) para calcular la primera componente. En su lugar, tendremos dos operaciones (parciales) para calcular la componente más a la izquierda cuando es un elemento básico (elemento-izp) y la componente más a la izquierda cuando es una lista general (lista-izq). Añadimos a las operaciones del enunciado una operación es-hsta-izq[7] que determina si la componente más a

la izquierda es una lista o no.

Con todas estas consideraciones, la signatura de la especificación queda como sigue, donde las operaciones restantes tienen el perfil esperado de su descripción en el enunciado. Hay además operaciones privadas que se explicarán más adelante, cuando se usen.

especificación *LISTAS-GENERALES[ELEM]* **usa** *BOOLEANOS, NATURALES* **tipos** *lista-gen* **operaciones**

operaciones privadas

variables

z : *elemento* x, y.

z : *lista-gen*

(constructora i

(constructora

}

(constructora

|

Como tenemos tres constructoras, en general distinguiremos tres casos en la definición ecuacional de las operaciones modificadoras y observadoras.

Como ya fiemos comentado antes, para calcular la componente más a la izquierda tenemos dos operaciones. La primera, que denominamos elemento-izq. solamente tiene sentido cuando la componente más a la izquierda es un

elemento básico, que es el resultado correspondiente, y da error en los otros dos casos. Por otra parte, la otra operación, que llamamos lista-izq. solo tiene sentido cuando la componente más a la izquierda es una lista general, que es el resultado correspondiente, y da error en los otros dos casos.

ecuaciones

elemento-izq([ ]) = error
elemento-izq(e : y) = *e* elemento-izq(x:: y) = error
lista-izq([ ]) = error
lista-izq(e : y) = error
lista-izq(x y) = x
es-lista-izq? ([ ]) = error
es-lista-izq?(e : y) = falso es-lista-izq?(x :: y) = cierto

En cambio, para eliminar la componente más a la izquierda, sea elemento básico o lista general, basta una única operación parcial, pues no tiene sentido en el caso vacío, y cuyo resultado en los otros dos casos es inmediato.

eliminar-izq([ ]) = error
eliminar-izq(e : y) = y
eliminar-izq(x :: y) = y

Una lista general puede no ser vacía y en cambio no tener ningún elemento básico, como por ejemplo la lista [[ ].[],[ ]] formada por tres listas vacías, o la lista unitaria [[|]]] que dentro tiene otra lista unitaria formada por la lista vacía. Por esta razón necesitamos una operación auxiliar hay-básicos? que nos indica si en una lista general

existen elementos básicos en algún nivel y sirve para caracterizar el caso en que las operaciones parciales básico-izq y básico-der no están definidas.

hay-básicos?([ ]) = falso

hay-básicos?(*e* : y)   = cierto

hay-básicos?(x :: y) = hay-básicos?(x) v hay-básicos?(y)

Para calcular el primer elemento básico en cualquier nivel, distinguimos casos sobre constructoras y usamos la operación auxiliar hay-básicos? que acabamos de ver para distinguir subcasos. El caso vacío está incluido en la primera ecuación que cubre el caso general de error. Cuando se añade un elemento básico, ese es claramente el básico más a la izquierda; cuando se añade una lista general, puede que en esa lista haya elementos básicos y entonces en ella estará el elemento buscado, o puede que no tenga y entonces tenemos que buscar en el resto de la lista.

básico-izq (x)
básico-izq(e : y)
básíco-izq(x :: y)
básico-izq (x ::

error <= -■hay-básicos?(x)

e

básico-izq(x) <= hay-básicos? (x)

básico-izq(y) <= -■hay-básicos?(x) A hay-básicos?(y)

La definición de básico-der se basa en las mismas ideas, con una distinción de casos adicional al añadir un elemento básico, que será el de más a la derecha cuando el resto de la lista no contenga

elementos básicos.

básico-der(x)	error <= -■hay-bás¡cos?(x)
básico-der(e : y)	*e* <= -■hay-básicos? (y)
básico-der(<?: y)	básico-der(y) <= hay-básicos?(y)
básico-der(x :: y)	básico-der(y) <= hay-básicos? (y)
básico-derU :: y)	básico-der(x) <= -■hay-básicos?(y) A hay-básicos?(.v)

Las definiciones de las operaciones para concatenar y calcular la longitud son como para las listas ordinarias (véase el Ejercicio 5.1), con la diferencia de tener que distinguir tres casos en lugar de dos, si bien los dos casos no vacíos se comportan de la misma forma.

[ ] ++ x
(e : y) ++ z
(x :: y) ++ z

e : (y -H- z)
-v :: (y ++ z)

longitud! [ ])
longitudes : y)
longitudU' :: y)

= 0
= I + longitud! v)
= I + longitud(y)

Para calcular ias inversas usamos operaciones auxiliares con acumulador (véase el Ejercicio 3.2), que se invocan originalmente con el acumulador vacío. La distinción de casos aparece entonces en la definición de la operación auxiliar.

En la primera operación, como en el caso de las listas ordinarias, en el caso vacío se devuelve el acumulador, y en los dos casos no vacíos la componente más a la izquierda del primer argumento pasa a ser la componente más a la izquierda del argumento acumulador, con lo que se consigue que todas las componentes del primer argumento se vayan colocando en el acumulador en el orden inverso.

Para la segunda operación hay que hacer lo mismo pero además, cuando la componente es una lista, también se tiene que invertir de la misma forma, por lo que se hace una llamada a la operación inversa-gen de manera que entonces las dos operaciones inversa-gen e inversa-gen-ac son mutuamente recursivas.

inversaLv) = inversa-aclx, [ ])
inversa-ac(| |. z) = z
inversa-ac(e : y. z) = inversa-ac(y, *e* : z)
inversa-ac(.v y. z) = inversa-ac(y. .v :: z)
inversa-gen(.\) = inversa-gen-ac(,r. [ ])

inversa-gen-ac([ ], z) = z
inversa-gen-ac(e *: v.* z) = inversa-gen-ac(y, *e* : z)
inversa-gen-ac(.v :: y, z) = inversa-gen-ac(y, inversa-gen(.v):: z)

El nivel de anidamiento de una lista general cuenta el número máximo de corchetes que pueden estar abiertos al ir entrando en listas anidadas. Por ejemplo, todas las listas ordinarias sobre naturales, incluyendo la vacía, tienen un nivel de anidamiento igual a L todas las listas de listas de naturales tienen un nivel de anidamiento igual a 2. etc. En las listas generales se mezclan listas con elementos, por lo que el nivel de anidamiento depende de cada lista.

anidamiento([|)
anidamientofi' : y)
an¡damiento(A :: v)

La dclinición ecuacional de la operación distingue los tres casos sobre constructoras. En el caso vacío el resultado es la constante I; cuando se añade un elemento, el nivel de anidamiento no cambia y coincide con el del resto de la lista; en tercer y último lugar, cuando se añade una lista, esta genera un nivel de anidamiento adicional, pero además hay que tener en cuenta el nivel que teníamos antes, por lo que hay que comparar el nivel de anidamiento del resto de la lista con el de la componente más a la izquierda incrementado en I, para quedarse con el máximo.

I
anidamiento(y)
máx(l + anidamiento(A), anidamiento!

v))

Para calcular la longitud máxima, a diferencia de la longitud normal que hemos visto más arriba, hay que tener en cuenta las longitudes de las listas que se añaden como componentes. Distinguimos los tres casos habituales. En el caso vacío el resultado es obviamente 0. Cuando se añade un elemento, tenemos que comparar la longitud máxima del resto de la lista, pues puede tener como componente en algún nivel una lista muy larga, con la longitud de la lista más externa que se incrementa en l al haber añadido un elemento, y quedarnos con el máximo de las dos posibilidades. Finalmente, cuando se añade una lista, comparamos la longitud máxima de la lista que se añade como componente más a la izquierda, la longitud máxima del resto de la lista, y la longitud de la lista más externa que se incrementa en l debido a la nueva componente que se ha añadido, y de nuevo nos quedamos con el máximo de las tres posibilidades.

longitud-máx([ ]) =0

longltud-máx(e : y) = máx(l + longitud(y), longitud-máx(y))

longitud-máx(.r :: y) = máx(longitud-máx(.r).máx(longitud-máx(y), l + longitud(v))) **fespecificación**

5.14. Diseñar una representación dinámica del TAD de las listas generales especificado en el Ejercicio 5.12 e implementar sus operaciones.

-------------------Solución------------------

En las listas generales especificadas solo se tiene acceso por el extremo izquierdo, salvo con la operación básico-der. pero este no tiene por qué ser el elemento más a la derecha de la lista. Por eso. a diferencia de la representación dada para las listas ordinarias (véase el Ejercicio 5.3). implementaremos las listas generales como una estructura enlazada con un único enlace al siguiente nodo. Una lista general será un enlace al nodo más a la izquierda. Esto hará que las listas generales siempre tengan que recorrerse desde la izquierda hacia la derecha.

Como un nodo de la lista puede almacenar un elemento básico o una lista general, utilizaremos un registro con un campo booleano *es-básico* y dos campos más: *elem* de Upo *elemento y lista* de tipo *lista- gen.* Dependiendo del valor de *es-básico* se utilizará uno u otro. El registro contiene además un enlace al siguiente nodo, como ya hemos indicado. Así, el tipo representante es el siguiente:

tipos

enlace-lista-gen **= puntero a** *nodo-lísta-gen*

nodo-lista-gen = **reg**

es-básico : bool

elem : elemento

lista : lista-gen
sig : enlace-lista-gen
freg
lista-gen = enlace-lista-gen
ftipos

La Figura 5.3 muestra una lista general utilizando esta representación. En la figura los valores boolea- nos cierto y falso se representan con C y F, respectivamente, y un guión - representa un campo indefinido. Las cuatro componentes de cada nodo corresponden a los cuatro campos del registro *nodo-lista-gen.*

La lista vacía no tiene ningún elemento, por lo que se representa con el enlace nulo.

fun lista-gen-vacía () **dev** *l : lista-gen* { 9(1))
/ := nil
ffun

Tenemos dos procedimientos para añadir elementos a una lista general, según el elemento sea básico o no. En cualquier caso, se reserva espacio para un nuevo nodo, se dice si el elemento que guarda es básico, y se enlaza con el resto de la lista.

proc añadir-elem(e *e : elemento. I : lista-gen)* { 0(1) }
var *p : enlace-lista-gen*
reservar(p)
*p*f*.es-básico* := cierto: *p^.elem := e*
p **t** *-sig := /*
/ := p
fproc

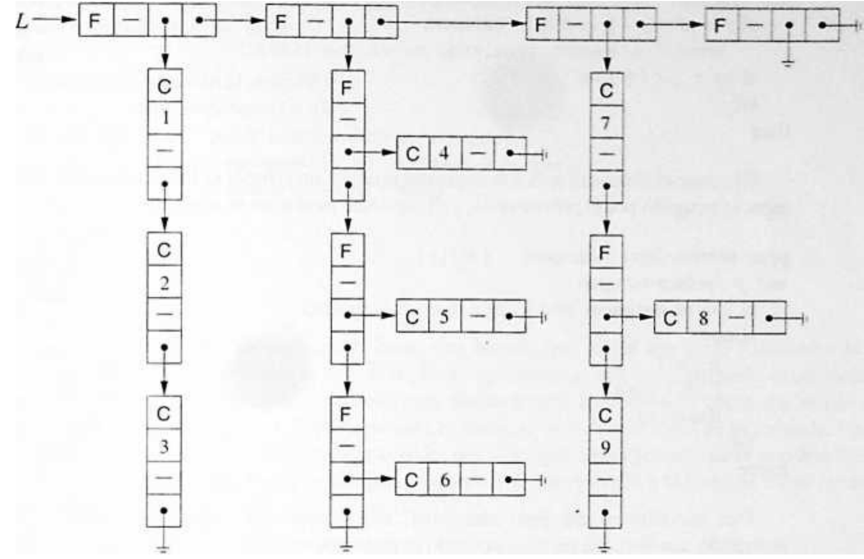


Figura 5.3: Representación dinámica de la lista general £ = [[1,2.3], [[4], [5], [6]]. [7. [8[. 9[. []].

proc añadir-lista-gen(e .r : *lista-gen, / : lista-gen')* (0(1)) **var** *p : enlace-lista-gen*

reservar)/?)

p f *.es-búsico* := falso ; *p* f *.lista :=* **x**

pT-s/g := /

/ := p

fproc

La función que comprueba si el elemento más a la izquierda es una lista es fácil de implementar.

fun es-lista-izq?(/ : *lista-gen)* **dev** *b : bool* [0(1))

si / = nil **entonces** error(Lista general vacía)

si no /? = -'(/ f *.es-biísico)*

fsi

ffun

También tenemos dos operaciones para consultar el elemento más a la izquierda, según este sea básico o no. Si la lista es vacía o contiene como elemento más a la izquierda uno de diferente tipo al buscado, se devuelve un error. En otro

caso, se devuelve ese elemento. Obsérvese el uso de v_c de forma que en el test se accede al elemento izquierdo solo cuando la lista no es vacía.

fun eiemento-izq(/ : *lista-gen)* **dev** *e : elemento* **(0(1))**
 si / = nil v_r -■(/ f *.es-biísico)* **entonces**
 error(El elemento izquierdo no es básico)
 si no *e* := / f *.elem*
 fsi
ffun

fun lista-izq(/ : *lista-gen)* **dev** x : *lista-gen* {<9(1)}
 si / = nil v_c *l* f *.es-básico* **entonces**
 error(El elemento izquierdo no es una lista)
 si no x *:=* *l* f *.lista*
 fsi
ffun

Eliminar el elemento más a la izquierda produce un error si la lista es vacía. En otro caso se libera el espacio ocupado por el primer nodo, y el siguiente pasa a ser el primero.

proc eliminar-izq(Z : *lista-gen)* (0(1) }
var *p : enlace-lista-gen*
 si *l* = nil **entonces** error(Lista general vacía)
 si no
 p ■*.= I*
 /:=/'!' .sig
 liberar(p)
 fsi
fproc

Para encontrar el elemento básico más a

la izquierda se recorre la lista con un enlace auxiliar *p,* utilizando una llamada recursiva cuando el nodo apuntado por *p* contiene una lista general. La función produce un error si la lista general global no tiene elementos básicos, pero las llamadas recursivas no tienen por qué producir el error. Por esa razón implementamos una función auxiliar más general que devuelve dos valores: un booleano que indica si la lista argumento tiene elementos básicos y. en caso de que este booleano sea cierto, el elemento más a la izquierda. El coste es lineal respecto a la suma total de nodos en todas las listas.

fun básico-izq(/ : *lista-gen)* dev *e : elemento*
{b.e) := básico-izq-aux(í)
si *-•b* **entonces** error(No hay básicos)
fsi
ffun
fun básico-izq-aux(/ : *lista-gen)* dev *{b : bool, e : elemento)*
var *p : enlace-lista-gen*
p := l; b := falso
mientras *p* / nil A *-'b* **hacer**
si pf.*es-básico* **entonces** *b* := cierto; *e := p^.elein*
si no *{b.e)* := básico-izq-aux(/ f *.lista)*
fsi
p := *p'f.sig*
fmientras
ffun

Debido a que solo tenemos un enlace al nodo izquierdo y luego enlaces a los

siguientes, para encontrar el elemento básico más a la derecha parece que hay que hacer un recorrido “desde la izquierda". Esto puede evitarse con un algoritmo recursivo que primero busca en el resto de la lista y que solo en el caso de que ahí no se haya encontrado, busca en el elemento más a la izquierda (con una llamada recursiva). Igual que antes, implementamos una función más general con dos resultados. El coste es lineal con respecto a la suma total de nodos en todas las listas.

fun básico-der(í : *lista-gen)* **dev** *e : elemento*
  {b.e) := básico-der-aux(í)
  si *-•b* **entonces** error(No hay básicos)
fsi
ffun
fun básico-der-aux(/ : *lista-gen)* **dev** *{b : bool. e : elemento)*
  si *I* = **mi entonces** *b :=* **falso**
  si no
    (buscamos antes en el resto de /)
    (b.e) := básico-der-aux(/ f *.sig)*
    si *-'b* **entonces {** puede estar en el primer nodo de /)
      si / f *.es-bcísico* **entonces** *(b.e) :=* (cierto. *I* f *.eleni)*
      sino *(b.e)* := básico-der-aux(/ f *.lista)*
      fsi
    fsi
  fsi
ffun
  La operación concatenar se implementa

como una función que recibe dos listas y devuelve la lista resultado de concatenarlas. Primero se copian las listas argumento usando una operación copiar-lista-gen que no detallamos y cuya implementación sería muy similar a otros algoritmos de copia que hemos dado (véase, por ejemplo, el Ejercicio 4.5). Si la primera es vacía, se devuelve la copia de la segunda. En otro caso, se recorre la primera lista hasta su último nodo, que se enlaza con el primero de la segunda lista. El coste es proporcional al coste de copiar las listas, que es lineal con respecto a la suma de todos los nodos en las dos listas.

fun concatenad v. y : *lista-gen)* **dev - :** *lista-gen*
var u. *w : lista-gen. p : enlace-lista-gen*
u := copiar-lista-gen(.v)
w := copiar-lista-gen(v)
si v = mi **entonces** z := *w* (el resultado comienza con *w)*
si no
| avanzamos hasta el último elemento de e)
I> := v
mientras *p* **j .sig nil hacer**
p := p'f.sig
fmientras
(y lo unimos con *w* |
p T ..víg : = *w*
z := r (el resultado comienza con u)
fsi ffun

Para calcular la longitud se recorre la lista sumando l por cada nodo enlazado.

El coste es lineal con respecto al número de nodos al primer nivel.

fun longitud!/ *- lista-gen)* **dev** *n : nat*
var *p : enlace-lista-gen*
n := 0 ; *p := /*
mientras *p* / nil **hacer**
n := n + l : *p := p* f *.sig*
fmientras
ffun

Para obtener la lista inversa, en el procedimiento inversa se recorren los nodos de la lista argumento desde el primero hasta el último, añadiendo cada uno por la izquierda a una lista que empieza siendo vacía. El coste es lineal con respecto a la longitud de la lista. Obtener la inversa general es muy similar, solo que ahora hay que invertir cada elemento que no sea básico, con una llamada recursiva. En este caso el coste es lineal con respecto a la suma total de nodos en todas las listas.

proc inversa(/ : *lista-gen)*
var *p, q. r : enlace-lista-gen*
p := l ; r := nil
mientras *p* nil **hacer**
{ añadir el nodo apuntado por *p* por la izquierda a *r)*
(sin perder el siguiente a *p)*
q ■- *P : p* := pt *sig*
q 1 *.sig := r; r := q*
fmientras
/ := r
fproc
proc inversa-gen(/ *: lista-gen)*
var *p.q.r : enlace-lista-gen*

p ;= I : r := nil
mientras *p* / nil **hacer**
? := P : P '•= P t sig
si -'(<? f *.es-básico)* **entonces** inversa-gen(r/ f *.lista)* **fsi**
q T *.sig* ■= *r ; r := q*
fmientras
i := **r**
fproc

Para implementar la función anidamiento recorremos los nodos de la lista manteniendo el máximo anidamiento encontrado. Cuando llegamos a un nodo que contiene a su vez una lista general, se calcula el anidamiento de esta con una llamada recursiva. El coste es lineal con respecto a la suma total de nodos en todas las listas.

fun anidamientof/ : *lista-gen)* **dev** *n : nat*
var *p : enlace-lista-gen*
p := l ; **n := 1**
mientras p nil **hacer**
si -■(pt *.es-básico)* **entonces** *n* :=. máx(n. 1 + anidamientolp f *.lisia))* **fsi**
P := pT-^g
fmientras
ffun

Una idea similar se sigue para implementar la función longitud-máx. Además de calcular el máximo de las longitudes máximas de las listas generales encontradas en nodos de la lista principal, hay que hacer el máximo con la longitud de dicha lista. Esta longitud se va calculando durante el

recorrido de la lista, y el máximo se hace al terminar. El coste es lineal con respecto a la suma total de nodos en todas las listas.

fun longitud-máx(/ : *lista-gen)* dev *n : nat*
 var *p : enlace-lista-gen*
  p := / ; n := 0 ; *long* := 0
  mientras *p* / nil **hacer**
   si *--(p j .es-básico)* **entonces** *n* := máxfn. Iongitud-máx(p f *.lisia))* **fsi**
   lons ⁱ= *long* + 1
   p := pt.sig
  fmientras
  n := máx(/i. *long)*
 ffun

aaaaaaaaaa

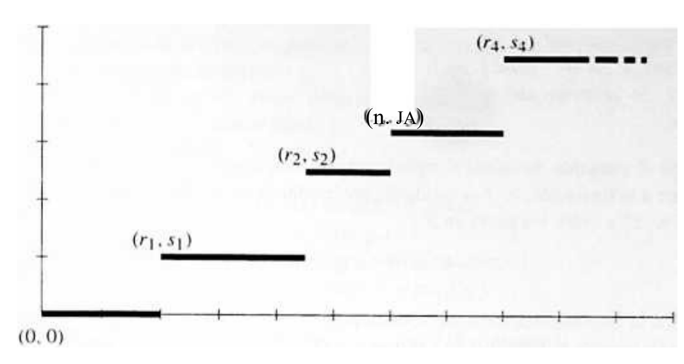


Figura 5.4: Función escalonada creciente

Una 5.14. función *f* : 1R$^+$ U {0) —> R es *escalonada* cuando consiste en una sucesión de funciones constantes definidas en subintervalos disjuntos, es decir, *f* puede definirse mediante condiciones de la forma /(.r) — *si* si *r,* < *x* < *ri+i.* Cada uno de los puntos en los que la función cambia de valor se llama un *salto.* Nos interesa la clase *de funciones escalonadas crecientes positivas* cuyo primer salto es (0. 0) y cuyo número de saltos es *finito.* Las funciones de esta clase tienen la forma mostrada en la Figura 5.4. Representar mediante listas las funciones escalonadas con las propiedades anteriores, e implementar las siguientes operaciones sobre tales funciones:

- crear la función constante nula,
- hacer una traslación de una función *w* unidades horizontalmente a la derecha y c unidades verti- caimente hacia arriba, con *w* > 0 y z > 0,
- calcular el valor de una función en un punto dado, y
- calcular la función máximo de dos funciones.

Solución:

Una función escalonada está determinada por el conjunto S de pares

tipos
par-real

función-escalonada
ftipos

(r,. si) que constituyen los saltos de la función. Como suponemos que el número de tales saltos es finito e incluye al menos el par (0. 0), vamos a representar una función escalonada como una lista no vacía de pares de números reales, que son los saltos correspondientes. Además, vamos a requerir que la lista de pares esté *ordenada* de menor a mayor según la coordenada .v, con lo que también estará ordenada de menor a mayor según la coordenada *y,* ya que la función es creciente.

reg
,v : *real*
y : real
freg
lista[par-real] (ordenada)

Para calcular los costes, vamos a suponer que los costes de las operaciones sobre listas son constantes, con la excepción de copiar, que es lineal con respecto a la longitud de la lista.

La función constante nula tiene únicamente el salto (0, 0), por lo que su implementación es inmediata.

fun constante-nula() **dev** *f : función-escalonada* { 0(l)) **var** *cero : par-real*
cero.x **:= 0 :** *cero.y* **:= 0**
f := unitaria(cero)
ffun

Si el conjunto de saltos 5 representa

una función escalonada *f.* el conjunto de saltos *S'* correspondiente a la traslación de *f w* unidades horizontalmente y z unidades verticalmente se obtiene sumando el par (tu, z) a todos los pares en *S:*

S' = ((r; + *w. si* + z) I i,) 6 S) U {(0, 0)).

En la implementación correspondiente no se hace nada cuando la traslación es la identidad. En otro caso se recorre la lista y cada salto se transforma sumando la traslación. La lista resultado se inicializa con el salto (0,0). En este algoritmo y los que siguen, como la lista se destruye al recorrerla, se hace una copia del argumento de entrada para no modificarlo. Debido a la copia y al recorrido, el coste es lineal con respecto a la longitud de la lista, es decir, al número de saltos de la función escalonada.

{ U' > 0 **A** z > 0)

fun trasladarf / : *función-escalonada, w.z : real)* dev *g : función-escalonada* **var** *h : función-escalonada, cero, par : par-real*

si *w* **> 0 v z > 0 entonces**

cero.x := 0 ; *cero.y :*= 0 ;

g := unitaria(cero)

h := copiar-lista(/)

mientras -^es-lista-vacía?(/i) **hacer**

par := izquierdo(/i) ; resto-der(fr)

par.x := par.x + w ; par.y := par.y + z añadir-der(g, *par)*

fmientras

fsi ffun

Para calcular el valor de una función escalonada en un punto dado, recorremos

la lista desde la izquierda buscando el salto apropiado al valor, que se encuentra al salir del bucle. Entonces basta devolver la coordenada y de ese salto como resultado. El coste es lineal con respecto al número de saltos, en todos los casos, debido a la copia de la lista argumento.
(w>0j
fun aplicad/ ; *función-escalonada, w : real)* dev z : *real*
var *h : función-escalonada, par : par-real*
  h := copiar-listaf/)
  mientras -'es-lista-vacía?(/i) A_r izquierdo(/i)..r < *w* hacer
    par := izquierdo(Zí) ; resto-der(/r)
  fmientras
  z *:= par.y*
ffun

Finalmente, para calcular la función escalonada resultante de hacer el máximo de dos funciones escalonadas, se recorren simultáneamente ambas funciones y se van comparando los saltos en la coordenada ,r; según el resultado de esa comparación, se coge el salto que va antes, en un proceso semejante a la mezcla de dos listas ordenadas (Ejercicio 5.7). Para determinar si ese salto se descarta o se añade a la función resultado, se compara su coordenada y con la del último salto añadido (por la derecha) al resultado, y solamente se añade cuando el valor de su coordenada y es mayor. Finalmente, cuando una lista se queda vacía hay que copiar los saltos restantes en la otra, siempre que superen

el valor del último salto añadido.
fun máx-función(*f. g : función-escalonada)* **dev** *h : función-escalonada*
var *f.* g' : *función-escalonada, último, parf, parg : par-real*
{ ponemos el primer salto del resultado)
últimos := 0 ; *último.y* := 0 ; *h* := unitaria(i<7fúno)
f := copiar-lista!/) : *g'* := copiar-lista(g)
(y los quitamos de los argumentos)
resto-der(/') . resto-der(g')
mientras -^es-lista-vacía?(/') A -■es-lista-vacía?(g') **hacer**
parf := izquierdo!/') : *parg* := izquierdo!.!»')
| cogemos los siguientes saltos de *f* y de *g* y los comparamos)
| ya sabemos que *último.x* < *patf.x* y *últimos* < *pargs)* **casos**
parf. v < *pargs* —► { el salto de *f* va antes y lo cogemos |
resto-der(/')
(lo comparamos con el último salto para decidir)
Mil := parf.y
si *val* > *último.y* **entonces**
últimos := parf.x . último.y := val
añadir-der(/i. *último)*
fsi
0 *parf .x = pargs* ->■ (los dos saltos coinciden en .r y los cogemos ambos) resto-der(/') ; resto-der(g')
| cogemos el mejor de los dos)
r«/ := *máxlparf.y. parg.y) (val >* *último.y* porque las funciones son

crecientes i *últimos := parf .x :*
último.y := val
añadir-der(/i, *último)*
□ *parf.x > pargs* —» (el salto de *g* va antes y lo cogemos)
resto-der(g')
(lo comparamos con el último salto para decidir }
val := parg.y
si *val > último.y* **entonces**
último.x := pargs ; últimos := val
añadir-der(/r, *último)*
fsi
fcasos
fmientras
(si g ya es vacía pero quedan saltos en *f)*
mientras ->es-lista-vacía?(/') **hacer**
parf := izquierdo!/') ; resto-der(/')
si *parf .y > últimos* **entonces**
{ copiamos todos los saltos de / que superen al último)
añadir-der(/i. *parf)*
fsi
fmientras
{ si *f* ya es vacía pero quedan saltos en *g)*
mientras -^es-lista-vacía?(g') **hacer**
parg := izquierdo(g') ; resto-der(g')
si *parg v > últimos* **entonces**
(copiamos todos los saltos de g que superen al último |
añadir-der(/r, *parg)*
fsi
fmientras
ffun

Como los dos recorridos se hacen simultáneamente, y en cada paso se considera uno de los saltos (a veces dos), el coste en tiempo es proporcional a la suma de las longitudes de las dos listas, es decir, a la suma de los números de saltos de las funciones escalonadas dadas como argumentos, que es también como mucho el número de saltos de la función máximo.

Capitulo 6

6. ARBOLES BINAROS Y GENERALES

Estructuras de datos organizados en forma de árbol aparecen por doquier al tratar toda clase de información: árboles genealógicos representando información familiar como los ascendientes o descendientes de una familia cualquiera o representando la historia de una dinastía real, árboles genealógicos de lenguas como la familia indoeuropea o de razas de animales, árboles sintácticos para analizar oraciones, árboles para representar la estructura anidada de los directorios y archivos en un disco duro, la clasificación de especies animales o. en general, la estructura jerárquica de cualquier organización, etc.

Las estructuras arbóreas que estudiamos en este capítulo generalizan las estructuras lineales vistas en capítulos anteriores, de forma que en vez de pasar de un dato al (único) siguiente, tenemos varios “siguientes'' que se consideran los *hijos* del dato en cuestión. Como en este caso, la terminología de las estructuras arbóreas se basa a veces en terminología genealógica ("padre”, "hijo", "primogénito", "siguiente hermano",

etc.), y en otras ocasiones en terminología botánica ("raíz", "hojas", "ramas", etc.), a la cual nosotros también añadiremos el término "plantar” para construir un árbol.

En los *árboles binarios* todo nodo tiene siempre dos hijos, aunque uno de ellos o ambos pueden ser vacíos. En particular, un árbol binario puede ser vacío. Los dos hijos se distinguen entre sí y se conocen como *hijo izquierdo* e *hijo derecho.*

En los *árboles generales* el número de hijos de cada nodo es variable, desde cero en el caso de una *hoja* hasta cierto número máximo que se llama el *grado* del árbol. Cuando el grado es n. también se dice que el árbol es n-ario, pero notemos que esto no significa que todos los nodos tienen exactamente *n* hijos, sino que *n* es el número máximo de hijos.

Es importante notar que los árboles binarios *no* son el caso particular de árboles n-arios con *n* = 2. En primer lugar, un árbol binario puede ser vacío mientras que un árbol general nunca es vacío. En segundo lugar, cuando un nodo en un árbol general tiene un único hijo, no se puede decir si es el izquierdo o el derecho porque esto no tiene sentido; en cambio, cuando un nodo en un árbol binario tiene un "único" hijo porque el otro es vacío, se puede distinguir si el hijo no vacío es el izquierdo o el derecho. Esto se ilustra en la Figura 6.1.
Aunque los ejemplos anteriores puedan dar la impresión de que los árboles

generales son más útiles que los binarios, de hecho, todo árbol general se puede representar mediante un árbol binario. Además, los árboles binarios adquieren especial importancia cuando tienen propiedades relacionadas con un orden sobre elementos, como veremos en los capítulos siguientes sobre árboles binarios de búsqueda (Capítulo 7) y montículos (Capítulo 8). Por otra parte, al tener una estructura con un número de hijos fijo y muy pequeño, los árboles binarios son más fáciles de comprender. Por todas estas razones, la mayor parte de los ejercicios que siguen están dedicados a los árboles binarios.

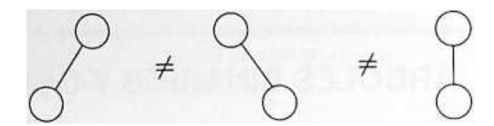


Figura 6.1: Dos árboles binarios y un árbol general con un único hijo.

Las operaciones sobre árboles, tanto generales como binarios, incluyen de forma destacada *recorridos* de diferentes clases. Para facilitar la realización de programas, los recorridos se especifican e implementan de forma que devuelven una lista en la que se acumula la información recogida en los nodos visitados a lo largo del recorrido, pero hay que entender que en general la “visita" a un nodo podría ser otra acción aparte de añadir información a una lista.

6.2. EJERCICIOS RESUELTOS

aaaaaaaaaa

6.1. Especificar un TAD para describir los *árboles binarios* cuyos nodos contienen elementos pertenecientes a un tipo dado corno parámetro, con las siguientes operaciones:

- crear el árbol vacío.
- construir un árbol a partir de un elemento y dos árboles.
- consultar la raíz,
- calcular el hijo izquierdo.
- calcular el hijo derecho, y
- determinar si un árbol es vacío.

-------------------Solución------------------

Como la estructura de los árboles es completamente independiente del tipo de los datos que se almacenan en los nodos, del cual no se requiere ninguna propiedad especial, la especificación es paramétrica con respecto al parámetro *ELEM,* visto en la Sección l. 1.5.

La elección de constructoras es obvia, pues cualquier árbol binario se construye de forma única partiendo del árbol vacío mediante la operación que añade un nodo raíz sobre dos árboles dados, que vamos a denominar plantar siguiendo con la analogía “botánica" de esta estructura. Estas dos constructoras son por tanto *libres.*

Las operaciones que consultan la raíz y los hijos no tienen sentido para el árbol vacío, por lo cual esas tres operaciones son parciales.

aaaaaaaaaa

especificación *ÁRBOLES-BINARIOS[ELEM]*
usa *BOOLEANOS*
tipos *árbol-bin*

árbol-vacío		*árbol-bin*	(constructora }
plantar	*árbol-bin elemento árbol-*	*árbol-bin*	(constructora }
raíz	*árbol-bin*	*elemento*	
hljo-iz	*árbol-bin*	*árbol-bin*	
hijo-dr	*árbol-bin*	*árbol-bin*	
es-árbol-vacío?	*árbol-bin*	*bool*	

variables

e : elemento

iz, dr : árbol-bin

Las operaciones raíz, hijo-iz e hijo-dr dan error en el caso vacío y son las "destructoras" asociadas a la constructora plantar en el caso no vacío, comportándose como proyecciones en tal caso, con lo cual la especificación ecuacional es muy sencilla, como se puede comprobar a continuación.

ecuaciones

hijo-iz(árbol-vacío) = error

hijo-iz(plantar(íc. *e. dr)) = iz*

hijo-dr(árbol-vacío) — error

hijo-dr(plantar(í;, *e, dr)) = dr*

raíz(árbol-vacío) = error

raíz(plantar(í'z. *e. dr)) = e*

es-árbol-vacío? (árbol-vacío) = cierto

es-árbol-vacio?(plantaríí;:. *e. dr))* = falso

fespecificación

Extender la especificación de los árboles binarios del Ejercicio 6.1 con las operaciones siguientes:

o calcular la altura de un tírbol. definida como el número de nodos de la rama más larga.

o calcular el número de nodos,

o calcular el número de hojas, es decir, nodos cuyos dos hijos son vacíos, y o calcular la imagen especular de un árbol.

-------------------Solución-----------------

Las cuatro operaciones son totales, por

lo que la signatura de la especificación queda como sigue.

especificación *ÁRBOLES-BINARIOS+[ELEM]*
usa *ÁRBOLES-BINARIOS[ELEMJ. NATURALES*
operaciones
alturaarbol-bin —> nat
núm-nodosárbol-bin —> nat
núm-hojas : *árbol-bin* —* *nat*
especulárárbol-bin —» árbol-bin
variables
e : elemento
iz. dr : árbol-bin

Todas las operaciones se definen distinguiendo casos sobre las constructoras. En el caso vacío, la altura y el número de nodos son ambos cero. Para calcular la altura en el caso no vacío, se calcula recursivamente la altura de los dos hijos y nos quedamos con el máximo; después añadimos 1. por la raíz.

El cálculo del número total de nodos es semejante, sumando los números de nodos de los dos hijos que se calculan recursivamente, y añadiendo 1 para contar también la raíz.

ecuaciones
altura! árbol-vacío) = 0
altura(plantar(í;. *e. dr))* = 1 + máxfaltura(ú). altura(dr))
núm-nodos(árbol-vacío) = 0
núm-nodos(plantar(íz. *e. dr))* = 1 + núm-nodos(íz) + núm-nodos(Jr)

Para calcular el número de hojas hay que considerar dos subcasos dentro del

caso no vacío, teniendo en cuenta que una hoja es un árbol no vacío cuyos dos hijos sí son vacíos. Así. en el caso vacío, el número de hojas es cero; en el caso no vacío, pero con hijos vacíos, el número de hojas es uno; y en el caso no vacío, con algún hijo no vacío, el número de hojas total se obtiene sumando los números de hojas de los dos hijos que se calculan recursivamente (pero nótese que ahora el nodo raíz no se cuenta porque no es una hoja, a diferencia del cálculo del número total de nodos).

núm-hojas(árbol-vacío) = 0

núm-hojas(plantar(íz. *e. dr))* — 1 <= es-árbol-vacío?(íz) A es-árbol-vacio?*(dr)*

núm-hojas(plantar(/z. *e, dr))* = núm-hojas(iz) + núm-hojas(rfr)

<= -'es-árbol-vacío?(rz) v -'es-árbol-vacío?(r/r)

La imagen especular de un árbol vacío es él mismo, y de un árbol no vacío se obtiene intercambiando los hijos izquierdo y derecho, y calculando recursivamente la imagen especular de ambos.

especular(árbol-vacío) = árbol-vacío

especular(plantar(íz. *e. dr))* = plantar(especular(r/r). *e.* especular(íz))

fespecificación

Diseñar una representación dinámica del TAD de los árboles binarios y desarrollar una implementación **de** las operaciones especificadas en el

Ejercicio 6.1.

-----------------Solución--------------------

La idea intuitiva es que la estructura enlazada corresponda a la representación gráfica de un árbol binario. De esta forma, cada nodo del árbol corresponde a un nodo en la estructura enlazada. Cada nodo necesita la información sobre el elemento que contiene y sobre la forma de acceder a los nodos "siguientes”. que en este caso corresponden a los dos nodos hijos en la estructura del árbol. Por tanto, un nodo de la estructura enlazada es un registro con un campo *valor* de tipo *elemento, y* dos campos tz, *dr* que son punteros señalando a los nodos siguientes, es decir, de tipo *enlace-árbol.* Un árbol se identifica con un enlace a su nodo raíz, siendo el árbol vacío representado con el puntero nulo. Un ejemplo de tal representación se muestra en la Figura 6.2. En resumen, el tipo representante es el siguiente:

tipos
enlace-árbol **= puntero a** *nodo-árbol*
nodo-árbol **= reg**
valor : elemento
iz. dr : enlace-árbol
freg
árbol-bin = enlace-árbol
ftipos

La implementación de todas las operaciones es inmediata, dado que el árbol vacío corresponde al puntero nulo, para construir un árbol basta crear un nodo que contenga en sus

diferentes campos la información de los argumentos (nótese que la estructura enlazada es compartida por el árbol que se ha creado y por sus hijos dados como argumentos), y para calcular la raíz y los hijos basta acceder a los correspondientes campos del nodo al que apunta el enlace que representa el árbol argumento.

fun árbol-vacío() dev *a : árbol-bin* { 0(1))

a ;= nil

ffun

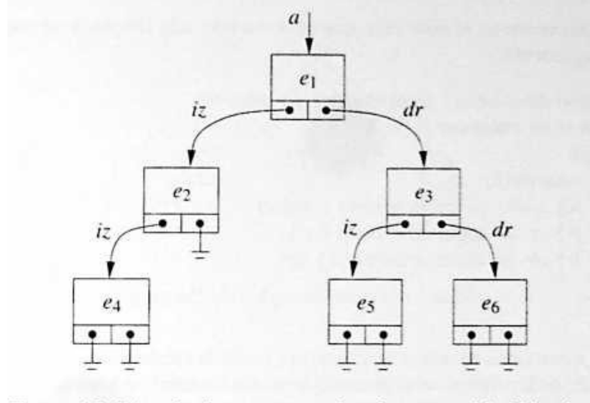


Figura 6.2: Ejemplo de estructura enlazada para un árbol binario.

fun plantar(/_- - *árbol-bin. e : elemento, dr : árbol-bin)* **dev** *a : árbol-bin* (0(1)) reservarlo) íí f *.valor := e a* f *.iz : = iz a* f *.dr := dr*
ffun
fun hi|o-iz(« : *árbol-bin)* **dev** *b : árbol-bin* (0(1) } **si** *a* = mi **entonces** error(Árbol vacío) **si no** *b : — a* j *.iz*
fsi
ffun
fun hijo-dr(o : *árbol-bin)* **dev** *b : árbol-bin* { 0(1)) **si** *a* = nil **entonces** error(Árbol vacío) **si no** *b := a* f *.dr*
fsi
ffun
fun raízlo . *árbol-bin)* **dev** *e : elemento* (0(1) } **si** *a* = nil **entonces** error(Árbol vacío) **si no** *e := a* f *.valor*
fsi
ffun
fun es-árbol-vacío? (« : *árbol-bin)* **dev** *b : bool* { 0(1)) *b := (a* = nil)
ffun

6.4. Utilizando la representación dinámica del Ejercicio 6.3. implementar operaciones para **copiar un árbol** binario y para liberar la

memoria ocupada por un árbol binario.

------------------Solución------------------

La operación copiar-árbol-bin tiene una implementación recursiva inmediata, siguiendo la estructura del árbol. El caso vacío es inmediato. En el caso no vacío, se copia usando la tunción apropiada según su

tipo el elemento en el nodo raíz, que es de un tipo más simple, y se copian recursivamente los hijos de árbol argumento.

fun copiar-árbol-bin(a : *árbol-bin)* **dev** *b : árbol-bin*
si a = nil **entonces** *b* := nil
si no
reservar(t)
b^.valor copiar-elem(rí f *.valor)*
b^.iz '■= copiar-árbol-bin(a f./z)
b f *.dr* ■*.*= copiar-árbol-bin(« f *.dr)*
fsi
ffun

La misma idea recursiva se aplica para anular la estructura. Cuando el árbol argumento no es vacío, s hacen llamadas recursivas al procedimiento anular-árbol-bin para anular los dos hijos, y después se anuí el nodo restante, usando para ello el procedimiento apropiado según el tipo del elemento en la raíz de árbol original.

proc anular-árbol-bin(o : *árbol-bin)*
si *a* / nil **entonces**
anular-árbol-bin(a f *.iz)*
anular-árbol-bin *(a* f *.dr)*
anular-elemfa f *.valor)*
liberaría)
fsi
fproc

Se dice que un árbol binario está *equilibrado* en altura si cumple las dos condiciones siguientes:

- la diferencia entre las alturas de sus

dos hijos es como mucho 1. y

· ambos hijos están también equilibrados en altura.

Demostrar mediante *inducción constructiva* que la altura de un árbol binario equilibrado de $n > 0$ nodos está acotada superiormente por 1 + clogn, para cierta constante real positiva *c*. Dar un contraejemplo que pruebe que necesariamente c > 1.

-------------------Solución----------------

Veamos primero el contraejemplo, para $n = 7$. Tenemos que el árbol de la Figura 6.3 tiene 7 nodo: es equilibrado, y su altura es 4 > I + log 7 = 3.8.

De hecho se cumple que la altura correspondiente es menor o igual que 1 + log n para todos lo árboles binarios equilibrados cuyo número de nodos es $n < 7$. corno se puede comprobar fácilment considerando todas las posibilidades.

Demostramos ahora el resultado que se pide en el enunciado. Sean *a* un árbol binario equilibrado, su altura, y n su número de nodos.

Como casos básicos consideramos $n = 1$, en cuyo caso $h = 1 = 1 + \log 1$, *y* $n = 2$ que da lugar $h = 2 = 1 + \log 2$, por lo que de momento bastaría tomar cualquier $c > 1$.

Sea ahora $n > 2$ *y* supongamos como hipótesis de inducción que el resultado es cierto para arbole binarios equilibrados no vacíos con menos de n nodos.

Si consideramos los dos hijos de *a y* les aplicamos la hipótesis de inducción (obsérvese que ell es posible, pues los hijos no pueden ser vacíos), tenemos *h*\ < 1 + clog/ii, /n < 1 + *c* logas, co *h* = máx(/ij. /is) + 1. *n* = ?u + ti2 + 1, y |/t| — /is| < I, por ser *a* equilibrado.

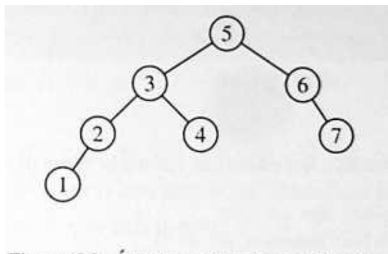


Figura 6.3: Árbol equilibrado con 7 nodos.

Tenemos que demostrar que *h* < I + *c* log/i, o lo que es lo mismo, que *n* > 2(A-l)/c

Tratamos en primer lugar el caso en que |/i| - /12I = 0. es decir. /i j = /n y por tanto *h* = /it + 1. Entonces

n — zii + «2 + l

>*h.i.* 7(/i|-l)/c j ot/n-lt/c | |

> 2</,|-1,/c + 2(Z,|_|,/C

_ —i+c)/e

> 2A1

como queremos demostrar, utilizando *c* > 1 en la última desigualdad y sin generarse ninguna restricción adicional sobre el valor de *c.*

Si |/i 1 - *h2* = 1. podemos suponer, sin pérdida de generalidad, que *h* > *I12* y por tanto /12 = /i| — 1 y *h* = *h* 1 + 1. Entonces

11 = n | +*112* + 1

7(/i|-t)/c j *llhz-Wc* . I

> 2(,I|-ri/c 2<A1-2)/c

_ 2(/'1-l)/<'(l +

>? 7(/í|-l)/col/c

—2A 1

_ 2<A-1>/c_

donde nos queda por demostrar la desigualdad señalada con interrogante, que es equivalente a

$1 + (2^{1/c})^{-1} - 2^{l/c} > 0.$

Para ello consideramos la función real

/(.v) = 1 +.r–| - x. cuya derivada es /'(x) = —1 — x~j. Como *f* es decreciente (puesto que /'-es negativa en todo su dominio) y tiene un único cero en x = ■ sabemos que / es positiva en (0. x) y negativa a partir de x. De aquí obtenemos que <■ debe satisfacer la desigualdad

,1/c < 1 +

— 2

de donde podemos despejar *c* para obtener la restricción explícita

c > zrr- 1.44,

única necesaria para que se cumpla el resultado buscado.

Escribir un algoritmo recursivo *eficiente* para comprobar si un árbol binario está equilibrado en altura, según la definición dada en el enunciado del Ejercicio 6.5.

Solución:

El algoritmo recursivo inmediato que calcula la altura de un árbol es el siguiente:

fun altura(« : *árbol-bin)* **dev** *alt: nat*
si es-árbol-vacío? («) entonces *alt* := 0
sino *alt* := 1 + máx(altura(hijo-iz(<?)). altura(hijo-dr(«)))
fsi
ffun

La recurrencia que describe el coste de altura es entonces:

T . , _ (c° *n* = O
+ *Tah(q) + C] n>0*

donde *n* es el número total de nodos de *a, p* y *q* son el número de nodos del hijo izquierdo y del hijo derecho de *a.* respectivamente, cumpliendo *p* -t- *q* = *n* - 1, y CQ *y ci* son constantes reales positivas.

Vamos a demostrar que el coste de altura es lineal con respecto al número de nodos, viendo que *Talt(n)* 6 ©(«)• Para ello vamos a ver en primer lugar que 7„/;(n) G 0(n). demostrando por *inducción constructiva* que

V/i : *n* > 1 : *Tnh(ii) < An*

para cierta constante real positiva A cuyo valor en términos de las

constantes co y cj se determinará a lo largo de la demostración.

En el caso básico *n* = 1 tenemos que

TaltiX) — Tjj/rtO) + *Ttjh(Q)* + ci = 2CQ + Cj < A.

de donde sacamos la primera restricción sobre la constante A, en relación con las constantes dadas CQ y C|.

Supongamos que la propiedad es cierta para cualquier *ni* < *n.* Entonces, en el paso de inducción tenemos para *n* > 1:

T«li(n) = Taitlpi + T_a*h(q)* + Cj <[1"1] *Ap* + *Aq* + Cj = *A(n* — I) + cj = *An* — A 4-C|.

De aquí.

A ti — A + cj ≤ *An* <=> — A - +- cj <0 <=> cj ≤ A.

que se cumple para cualquier constante A que cumpla la primera restricción porque ci < 2CQ + CI < A.

Por consiguiente, basta tomar A = 2CQ + c, para que en efecto Vn : *n* > 1 : T_a*i,(n)* < *An.*

En segundo lugar, vamos a ver ahora que T_a*u* (n) G Q *(n*). demostrando mediante *inducción constructiva* que *VH : n* > 0 : T_a*i,(n)* > *Bn*

para cierta constante real positiva *B* cuyo valor en términos de las constantes dadas CQ y c, se determinará a lo largo de la demostración.

En el caso básico *n* = 0 tenemos que 7flh(0) = co > 0,

que se cumple independientemente del valor de *B*.

Supongamos que la propiedad es cierta para cualquier *ni* < *n.* Entonces, en el paso de inducción tenemos para *n* > 0:

Tal,(n) = T_a*i,(p)* 4- T_{ah}*(q)* 4-cj >'" *Bp* + *Bq* 4-C| = *B(n* **- 1)** 4- **c, =** *Bn* - *B* 4- **C|.**

De aquí,

Bn — *B* 4- C| > *Bn* <=> —*B* 4* ci > **0** s=> > *B.*

de donde sacamos una restricción sobre la constante *B.* en relación con la constante C|.

Por consiguiente, basta tomar *B* = *c* para que Vn : n > 0 : T_a*i*$_t$*(n)* > *Bn.*

Habiendo justificado el coste lineal de altura, pasamos a desarrollar una función que comprueba si un árbol binario está equilibrado, implementando directamente la definición de árbol equilibrado en el enunciado del Ejercicio 6.5. Para ello usamos una función diferencia que calcula la diferencia entre dos números naturales de forma que siempre se resta el número menor de los dos.

fun diferencia(/i. *ni* : nat) **dev** *d : nat*
si *n* > *ni* **entonces** *d* := *n* — *ni*
si no *d* := *ni* — *n*
fsi
ffun
fun equilibradoto : *árbol-bin)* **dev** *eq : bool*
si es-árbol-vacío?(«) **entonces** *eq* := cierto

si no *eq .*= equilibrado!hijo-iz(a)) A equilibrado(hijo-dr(n)) A
(diferencia(altura(hijo-iz(ri)).
altura(hijo-dr(«))) < 1)
fsi
ffun

La recurrencia que describe el coste de la función equilibrado es la siguiente, teniendo en cuenta que las llamadas que se hacen a la función altura tienen un coste lineal con respecto al número de nodos:

T. (ni = ! CÓ " = o
′ | $T_{eq}(p) + T_{eq}(q) + c_t n$ *n >0*

donde *n. p* y *q* son como en la recurrencia anterior, y *c'„ y* cj son constantes reales positivas.

Vamos a ver en primer lugar que esta recurrencia no determina en general una única función debido a las elecciones de *p* y *q.*

Cuando el árbol se divide precisamente de forma "equilibrada", con aproximadamente la mitad de nodos en cada hijo partí cada llamada recursiva, la recurrencia se aproxima de la forma

T í có " — o
r(,,) ~ | *2T(n/2) + c\n n* **> 0**

cuya solución está en *&(n* log *n)* por el teorema de la división para recurrencias básicas (véase por ejemplo [Peñ98, Capítulo 1]).

Cuando el árbol es degenerado y se comporta como una lista, uno de los hijos es siempre vacío, de forma que la recurrencia queda

1 **^{c}ó** *n = 0*
T(n - 1) 4-c¿4-<'í« *n* > 0
cuya solución está en $0(n^2)$ por el teorema de la resta para recurrencias básicas [Peñ98. Capítulo 1].

Pero esto no prueba que no haya casos peores todavía con otras elecciones de *p* y *q.* Para descartar esta posibilidad, veamos que 7j.?(n) 6 *O(ir)* (para cualquier elección de *p* y *q).* demostrando mediante *inducción constructiva* que

Vn : *n* > 1 : T^ín) < *Cn~*

para cierta constante real positiva *C* cuyo valor en términos de las constantes dadas *c'o y* cj se determinará a lo largo de la demostración.

En el caso básico $n = 1$ tenemos que r_{f9}(l) = r«/(0) + T_r,(0) + cí = *2c'o* + c'j < C.

de donde sacamos la primera restricción sobre la constante C, en relación con las constantes *Cg y* c'|.

Supongamos que la propiedad es cierta para cualquier $m < n$. Entonces, en el paso de inducción tenemos para $n > 1$:

$$T_{eq}(n) = T_{.,q}(p) + T_{eq}(q) + c'pi$$
$$Cp^2 + Cq^2 + c\backslash n.$$

Como $p + q = n - 1$, $p^2 + q^2 = (n — l)^2 - 2pq < (n — l)^2$. por lo que
$T_{fí}(n) < C(n — l)^2 + cjn = Cn^2 — 2Cn + C + c\backslash n.$

De aquí,

$$Cn^2 — 2Cn + C + c\backslash n < Cn^2 <=>$$
$$C + (c, — 2C)n < 0.$$

De acuerdo con la primera restricción, podemos tomar como constante $C =$ *2c'o* + cj. y comprobamot que esta elección satisface también la última restricción:

$$(2cQ + c,) + (c'| - 2(2c'_o + C|))n =$$
$$(2c¿ - 4c¿n) + (c, - c,n) <0 + 0 =$$
$$0.$$

porque $n > 1$.

Por consiguiente, basta tomar C = *2c'o* + c, para que en efecto Vn : n > 1 : $T_{cq}(n) < Cn$~

En resumen, el coste en el caso peor de la función equilibrado es cuadrático con respecto al número dt nodos del árbol argumento. Aunque evidentemente correcta, esta función es poco eficiente debido a qui la altura de los subárboles se calcula demasiadas veces, haciendo repetidos recorridos de los subárboles Por ejemplo, la llamada recursiva equilibrado(hijo-iz(«)) necesita calcular las alturas de sus hijos, perc después. altura(hijo-iz(«)) vuelve a calcularlas.

La solución que evita esas innecesarias repeticiones consiste en ir determinando *simultáneamente* s el árbol está equilibrado y cuál es su altura de forma que. intuitivamente, se haga un único recorrido de árbol. El algoritmo queda entonces como sigue:

fun equilibrado2(« : *árbol-bin)* dev *(eq : bool.alt*: nat)

si es-árbol-vacío?(a) entonces *(eq.alt)* := (cierto. 0)

si no

(eq-iz. alt-iz) := equilibrado2(hijo-iz(n))

(eq-dr, alt-dr) := equilibrado2(hijo-dr(«))

eq := eq-iz A *eq-dr* A (diferencia(a/r-ú, *alt-dr)* < 1)

alt := 1 + máx(a/r-íz. *alt-dr)*

fsi

ffun

El coste del nuevo algoritmo equilibrado2 se describe mediante la

recurrencia

т, í ᶜo *n = 0*

| $T_{eq}2(p) + T_{eq}2(q) + c''$ $n>0$

que tiene exactamente la misma forma que la del algoritmo altura vista al principio y cuya solució: está por tanto en 0(n), como ya hemos demostrado. Así pues, el coste del nuevo algoritmo es linea con respecto al tamaño del árbol, en vez de cuadrático como el algoritmo que seguía directamente I definición y, por tanto, el nuevo algoritmo es mucho más eficiente.

6.7. Se dice que un árbol binario es ***zurdo*** si o bien es el árbol vacío o una hoja, o bien sus hijos izquierdo y derecho son ambos zurdos y más de la mitad de sus descendientes están en el hijo izquierdo. Desarrollar un algoritmo recursivo de coste lineal para decidir si un árbol binario es zurdo.

-----------------Solución----------------

Si el árbol es vacío o es una hoja, ya sabemos por la definición que es zurdo. En otro caso, para comprobar si el árbol es zurdo, tenemos que comprobar primero que sus hijos sean zurdos. Si los dos hijos son zurdos, tendremos que comprobar, además, si hay más descendientes izquierdos que derechos, ya que, si *di* es el número de descendientes izquierdos y *dd* es el número de descendientes derechos.

la mitad de los descendientes es $^{di+dd}$ y el árbol es zurdo si

, di + dd di > ——— <=> 2di > di + dd <=> di > dd.

2

Aplicando la misma idea que en la solución del Ejercicio 6.6 para no hacer cálculos repetidos, es claro que resulta conveniente que las llamadas recursivas con los hijos izquierdo y derecho devuelvan también el número de nodos en cada hijo, de forma que el algoritmo determine *simultáneamente* si el árbol es zurdo y cuál es su número de nodos, haciendo así un único recorrido del árbol. El algoritmo resultante es entonces el

siguiente:

fun zurdoU/ : *árbol-bin)* **dev** *{zurdo : bool.nuni : nat)* **casos** es-árbol-vacío?(o) —► *{zurdo, num}* := (cierto. 0)

▫ es-hoja?(rr) —> *(zurdo, num*) := (cierto. 1 }

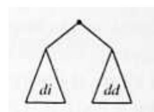
▫ ~'es-árbol-vacío?(«) A -’es-hoja?(o) —>

(zur-iz, des-iz} := zurdoíhijo-iz(rí)) *{zur-dr, des-dr}* : = zurdo(hijo-dr(«)) *zurdo := zur-iz r\ zur-dr {des-iz > des-dr) num* := 1 + *des-iz* + *des-dr* **fcasos ffun**

El predicado es-hoja?(«) que se utiliza en la distinción de casos del algoritmo anterior es equivalente a -’es-árbol-vacío?(<i) A es-árbol-vacío?(hijo-iz(«)) A es-árbol-vacío?(hijo-dr(«)).

El coste del algoritmo zurdo está en 0(n), siendo *n* el número de nodos del árbol *a.* ya que solo pasamos una vez por cada nodo. Para ello, basta comprobar que la recurrencia que describe el coste de zurdo es la misma que la asociada a los algoritmos altura y equilibrado2 en la solución del Ejercicio 6.6.

6.8. Dado un árbol, la *distancia* entre dos



nodos es la longitud del único camino que los conecta, y el *diámetro* del árbol es la distancia máxima sobre todos los pares de nodos. Desarrollar un algoritmo de coste lineal (con respecto al número de nodos del árbol) para hallar el diámetro de un árbol binario dado.

------------Solución-----------------

Si los dos nodos sobre los que se obtiene la distancia máxima están en el mismo hijo, el diámetro del árbol padre coincide con el diámetro del hijo correspondiente, por lo que hay que hacer llamadas recursivas con los hijos.

Si cada nodo está en un hijo diferente (con respecto al árbol original), el camino que los une pasa necesariamente por la raíz, y en este caso, para que la distancia sea máxima, esos nodos tienen que estar lo más lejos posible de la raíz, por lo que la distancia entre ellos coincide con la suma de las alturas de los dos hijos. Nótese que la altura de un árbol cuenta el número de niveles del árbol, es decir, cuenta los nodos por los que se va pasando, mientras que la distancia entre dos nodos cuenta el número de aristas que los une.

En general, como no sabemos cuál de los tres casos se va a dar, calculamos

las tres posibilidades y nos quedamos con la mejor, que es el máximo de las tres.

Ahora bien, si se hacen llamadas a una función altura de coste lineal, el coste final del algoritmo que se pide será cuadrático con respecto al número de nodos del árbol. Para mejorar la eficiencia y obtener un coste lineal, hay que calcular la altura al mismo tiempo que el diámetro, siguiendo la idea descrita en las soluciones de los Ejercicios 6.6 y 6.7.

No hace falta distinguir el caso de una hoja, y basta como caso básico el del árbol vacío, si tenemos un poco de cuidado utilizando el valor — 1 para el diámetro de un árbol vacío y el valor 0 para el diámetro de un árbol que sea una hoja con un único nodo.

fun diámetro(<i : *árbol-bin)* **dev** *(didin : ent. alt* : nat)

si es-árbol-vacío?(a) entonces *(diám.alt) :=* (—1.0)

si no

{diám-iz,alt-iz) := diámetro(hijo-iz(«))

(diám-dr. alt-dr) := diámetro(hijo-dr(«))

dichn máx(máx(rfiám-íz. *diám-dr), alt-iz* + *alt-dr)*

alt 1 + máx(a//-íz, *alt-dr)*

fsi

ffun

Este algoritmo tiene un coste en *Q(n).* donde *n* es el número de nodos del

árbol *a.* pues la recurrencia que describe su coste es la misma que la de los algoritmos altura y equilibrado2 en la solución del Ejercicio 6.6.

6.9. Especificar e implementar de forma abstracta los tres *recorridos en profundidad* de árboles binarios: *preorden, inorden* y *postorden.*

-----------------Solución------------------

La especificación de los recorridos está parametrizada, igual que la de los árboles binarios, con respecto al parámetro *ELEM.* El resultado de cada recorrido es una lista de elementos sobre el mismo tipo que el árbol, y los tres recorridos son operaciones totales.

especificación *RECORRIDOS-PROF-ÁRBOLES-BINARIOS[ELEM]*
usa *ÁRBOLES-BINARIOSfELEM], LISTAS[ELEM]*
operaciones
preorden : *árbol-bin —> lista*
inorden : *árbol-bin lista*
postorden: *árbol-bin —> lista*
variables
e : elemento
iz, dr : árbol-bin

En el caso vacío, los tres recorridos estándar de árboles binarios devuelven la lista vacía. En el caso no vacío, se caracterizan por recorrer recursivamente el hijo izquierdo y el hijo derecho (siempre en ese orden) y añadir la raíz. El recorrido en *preorden* la añade al principio, el recorrido en *inorden* (también conocido como simétrico) la añade entre los recorridos de los dos hijos, y el recorrido en *postorden* la añade al final.

Escribimos todas las ecuaciones usando

la operación de concatenación para destacar **el parecido** entre ellas, pero también se podrían utilizar otras operaciones sobre listas para escribir la parte derecha de las ecuaciones, como para añadir un elemento por la izquierda o para añadir un elemento por la derecha.

ecuaciones

preorden(árbol-vacío) = []

preorden (plantar!/’, *e, dr))* = [e] -H- (preorden!/;) -H- preordenlr/r))

inorden(árbol-vacío) = [ ]

inorden! plantar!/’. *e. dr)) =* inorden(íc)-H-([e]-H-inorden(Jr))

postorden(árbol-vacío) = [ ]

postorden(plantar(/’. *e. dr)) =* postorden(íz) ++ (postorden(r/r) -H- [e)) **fespecificación**

La implementación abstracta de los tres recorridos sigue exactamente la misma idea que la definición ecuacional anterior; la única diferencia es la notación algorítmica para distinguir casos y hacer llamadas recursivas, y el uso apropiado según sean funciones o procedimientos de los algoritmos que implementan las operaciones sobre listas.

fun preorden!// *. árbol-bin)* **dev / ;** *lista* **var** *r : lista*

si es-árbol-vacío?(c/) **entonces / :=** lista-vacía()

si no

r := preorden(hijo-iz(o))

añadir-izq(raíz(r/), *r)*

I := concatenar!/'. preorden(hijo-dr(o)))
fsi
ffun
fun inorden!// . *árbol-bin)* **dev /:** *lista*
var /' *: lista*
si es-árbol-vacío?(«) **entonces** *I* := lista-vacía!)
si no
/■ := inorden(hijo-iz(«))
añadir-der(/-. raiz(o))
I := concatenar!/-, inorden(hijo-dr(«)))
fsi
ffun
fun postorden(// ; *árbol-bin)* **dev** *I : lista* **var /' :** *lista*
si es-árbol-vacío⁹!//) **entonces / :=** lista-vacía!)
si no
/■ := postorden(hijo-iz(í/))
/ := concatenar(r, postorden(hijo-dr(//)))
añadir-der(/, raiz(o))
fsi
ffun

Suponiendo que las operaciones básicas sobre listas y sobre árboles que se usan en estos tres algoritmos están implementadas con coste constante, la recurrencia que describe el coste de cada uno de los tres algoritmos anteriores tiene la forma

T(n) = co 'i = 0
T(i>) + *T(q)* + c, //> 0 donde n es el número total de nodos del árbol *a. p* y <7

son los números de nodos de su hijo izquierdo y derecho, cumpliendo $p + q = n - 1$. y CQ y ci son constantes reales positivas.

Como demostramos con todo detalle en la solución del Ejercicio 6.6. *T(n)* e ©(n). por lo que el coste de cada recorrido es lineal con respecto al número de nodos del árbol que se recorre.

El coste en espacio adicional es también lineal debido a la lista auxiliar que hace falta declarar para poder invocar el procedimiento apropiado en cada caso.

6.10. (a) Desarrollar un algoritmo que reconstruya un árbol binario a partir de las dos listas que definen sus recorridos en preorden e inorden, sabiendo que todos los elementos (en el árbol, y por tanto en cada una de las listas) son distintos.

(b) ¿Es posible reconstruir el árbol unívocamente cuando las dos listas dadas son sus recorridos en inorden y postorden?

(c) ¿Y si las dos listas son sus recorridos en preorden y postorden?

------------------Solución-------------------

Apartado (a)

Sean *Ipre* y *Un* las listas que contienen los recorridos en preorden e inorden, respectivamente, de un árbol binario. Claramente, las dos listas tienen los mismos elementos, pero en diferente orden El árbol es vacío si y solo si las dos

listas lo son. Por tanto, en la discusión que sigue solamente consideramos el caso no vacío.

El recorrido en preorden visita primero la raíz del árbol, y luego los hijos izquierdo y derecho recursivamente. Por tanto, el primer elemento de la lista *Ipre,* al que denotaremos por v. es la raíz del árbol buscado.

En el recorrido en inorden, primero se visita el hijo izquierdo recursivamente, después la raíz del árbol, y finalmente se visita el hijo derecho recursivamente. Así el elemento v aparecerá en la lista *Un* en cierta posición *k.* Todos los elementos en la lista *Un* anteriores a esa posición pertenecen al hijo izquierdo del árbol buscado y todos los posteriores al hijo derecho. Es más, una vez conocemos el valor de *k.* los elementos en la lista *Ipre* entre las posiciones 2 y *k* nos dan el recorrido en preorden del hijo izquierdo y los siguientes el del hijo derecho. Conociendo estos datos, podemos construir los hijos izquierdo y derecho del árbol buscado haciendo sendas llamadas recursivas sobre las sablistas correspondientes.

En el algoritmo que sigue, además de operaciones usuales sobre listas, vamos a usar la operación posición, especificada en el Ejercicio 5.2 e implementada como función en el Ejercicio 5.5. que calcula la posición de un elemento en una lista, y las operaciones coger y tirar, especificadas e implementadas como

procedimientos en el Ejercicio 5.6. que "cogen" y "tiran" respectivamente cierto número de elementos del comienzo de una lista. Como estos procedimientos destruyen parte de la lista argumento, es necesario copiarla antes de llamarlos.

fun reconstruir-árboll *(Ipre. Un : lista)*
dev *a : árbol-bin*
var íz. *dr : árbol-bin. Ipre-iz, Ipre-dr. Un-iz. Un-dr : lista*
si es-lista-vacía?*(Ipre)* entonces *u* := árbol-vacío()
si no
x := izquierdo(//jre)
k := posición(.r, *Un)*
Ipre-iz := copiar-lista(//?re) ; elim-izq(//w-íz) ; coger(¿ - 1, *Ipre-iz)*
Un-iz := copiar-lista(/úi) : coger(Zr - *Un-iz)*
iz := reconstruir-árbol 1 *(Ipre-iz. Un-iz)*
Ipre-dr := copiar-lista(//?re) ; tirar (ir, *Ipre-dr)*
Un-dr := copiar-lista(/úr) : tirar(¿. *Un-dr)*
dr := reconstruir-árboll *(Ipre-dr. lin-dr)*
a := plantar(rz, x, *dr)*
anular-lista(/pre-íz) ; anular-lista(//rre-</r)
anular-lista(//7i-í'z) : anular-lista(/úi-r/r)
fsi ffun

Aunque algunas operaciones básicas sobre listas y árboles estén implementadas con coste constante,

operaciones como posición, coger y tirar van a tener coste lineal en el caso peor (con respecto al tamaño de las dos listas dadas como argumento, que tienen la misma longitud). Por tanto, la recurrencia que describe el coste en el caso peor de reconstruir-árboll tiene la forma

T(n) = I Co n = 0

| *T(p)*+ *T(q)* + *c\n n>0*

donde *n* es la longitud de las dos listas, *p* y *q* son las longitudes de las sublistas resultantes del proceso de descomposición, cumpliendo *p* + *q* = *n* — l. y CQ y ci son constantes reales positivas.

Como demostramos en la solución del Ejercicio 6.6. en el caso peor *T(n)* e ©(«*), por lo que el coste de la reconstrucción del árbol es cuadrático con respecto a su número de nodos.

El coste en espacio adicional es cuadrático debido a que se hacen copias en variables auxiliares de tipo *lista* en todas las llamadas recursivas.

Apartado (b)---------------------

Dados los recorridos en postorden e inorden, *Ipost y luí.* la idea es completamente análoga, pero buscando la raíz al final de la lista *Ipost,* por lo que hace falta utilizar las operaciones sobre listas derecho y elim-der El correspondiente algoritmo queda como sigue:

fun reconstruir-árbol2(/p<«7. *lin : lista)*

dev *a : árbol-bin*
 var i.⁻, *dr . arbol-bin. lpost-iz, Ipost-dr, lin-iz, lin-dr : lista*
   si es-lista-vacía?(//?o.s7) **entonces** *o* árbol-vacío() **si no**
     A = derecho(//w.s7)
     k posición(A . *luí)*
     lpost-iz := copiar-hsta(//;o.s7) : coger(< - \, *lpost-iz)*
     lui-iz '.= copiar-lista(/m) : cogerU — *lin-iz)*
     iz = reconstruir-árbol2(/po.s7-i;. *lin-iz)*
     Ipost-dr .= copiar-lista(//?os7) ; *e\\m-der(lpost-dr) ;* tirar(A‘ - l. *Ipost-dr)*
     lin-dr := copiar-lista(*lin) ;* tirar(Á:. *lin-dr)*
     tlr := reconstruir-árbol2(/post-r/r. *lin-dr)*
     a = plantar^;, A , *dr)*
     anular-lista(//wsr-r;) ; anular-lista(//?ru7-r/r)
     anular-lista(/úi-ic) : anular-lista(/úi-r/r)
   fsi
 ffun

El análisis de costes en el apartado anterior también es aplicable a este algoritmo.

Apartado (c)----------------------------

En cambio, si tenemos los recorridos en preorden y postorden, el árbol no está determinado de forma unívoca. Por ejemplo, los dos árboles de la Figura 6.4 son distintos, pero tienen los mismos recorridos en preorden [3,2, l] y

postorden [1,2, 3[. El recorrido en inorden, por supuesto, sí es diferente: [l. 2. 3[para el árbol de la izquierda y [3, 2, 11 para el de la derecha.

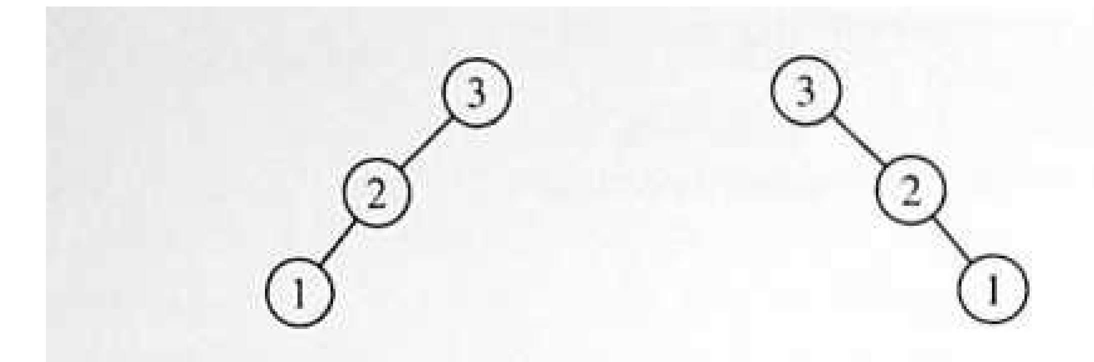


Figura 6.4: Dos árboles con igual recorrido en preorden y postorden.

6.11. El objetivo final de este ejercicio es obtener una versión iterativa del recorrido en preorden de un árbol binario. Para ello hay que obtener una generalización adecuada de la función preorden, la operación pre-lista-pila que. dadas una *lista* y una *pila de árboles binarios,* devuelve la concatenación de la lista con la concatenación de los recorridos en preorden de todos los árboles en la pila.

(a) Definir una nueva operación pre-pila sobre *pilas de árboles binarios,* que devuelve la concatenación de los recorridos en preorden de todos los árboles en la pila, empezando por la cima. Escribir las ecuaciones de su especificación en términos de preorden.

(b) Escribir la especificación de la operación pre-lista-pila en términos de pre-pila.

(c) Operar ecuacionalmente utilizando las ecuaciones de las dos definiciones anteriores junto con las ecuaciones recursivas que definen el recorrido en preorden de un árbol

binario (véase el Ejercicio 6.9). para obtener ecuaciones recursivas que definen la operación pre-lista-pila solo en términos de sí misma, sin utilizar pre-pila ni preorden.

(d) Utilizar esa definición recursiva para obtener la versión iterativa de pre-lista-pila.

(e) A partir de ese algoritmo, escribir la versión iterativa del algoritmo que calcula el recorrido en preorden de un árbol binario.

--------------------Solución--------------

Apartados (a) y (b)

Las ecuaciones para las dos nuevas operaciones son inmediatas, utilizando la operación de concatenación sobre listas.

pre-pila(pila-vacía) = []

pre-pila(apilar(«, p)) = preorden(o) 44-pre-pila(p)

pre-lista-pila(/, *p)* = *I* 44- pre-pila(p)

De esta forma, para calcular el recorrido en preorden de un árbol binario basta considerar la siguiente ecuación, que nos dice cómo debemos inicializar los parámetros auxiliares de la generalización (la lista y la pila):

preorden(rz) = pre-lista-pila([ ]. apilar(o. pila-vacía))

Recordemos que las ecuaciones recursivas que definen el recorrido en preorden de un árbol binario son las siguientes.

preorden(árbol-vacío) = | |

preorden(plantar(/z. *e. dr))* = |e] 44-(preorden(íz) 4+ preorden(dr))

Apartado (c)-----------------

Para obtener la definición recursiva de pre-lista-pila. vamos a operar ecuacionalmente con las ecuaciones anteriores (salvo la cuarta), distinguiendo los tres casos siguientes (no hace falta distinguir casos sobre la lista):

- La pila es vacía.

  pre-lista-pila(/, pila-vacía)
  = / 44- pre-pila(pila-vacía)
  = /4+[]
  = /

- La pila no es vacía, pero el árbol en su cima es el árbol vacío.

  pre-lista-pila(/. apilar(árbol-vacío. p))
  = / 44- pre-pila(apilar(árbol-vacío. p))
  = /44- (preorden(árbol-vacío) 44- pre-pila(p))
  = / -H- ([ ] -H- pre-pila(p))
  = / 4+ pre-pila(p)
  = pre-lista-pila(/. p)

  Claramente en este caso la recursión está bien definida porque el tamaño de la pila de árboles disminuye estrictamente.

e La pila no es vacía y el árbol en su cima no es el árbol vacío.

  pre-lista-pila(/. apilar(plantar(í'c. *e. dr).* p))
  = / 4+ pre-pila(apilar(plantar(í;. *e. dr). p))*
  = *l* 44- (preorden(plantar(í;, *e, dr))* 44- pre-pila(p))
  = *l* 44- (([<’] 44- (preorden(í-) 44-

preorden(dr))) 44- pre-pila(p))
= (/ 44- [<’]) 44- (preorden(íc) 44- (preorden(íZr) 44- pre-pila(p)))
= (/ 44- [e]) 44- (preorden(íc) 44- pre-pila(apilar(</r. p)))
= (/ 44- [<?]) 44- pre-pila(apilar(íc. apílar(r/r, p))>
= pre-lista-pila(/ 44- [<?], apiladle. apilar(i/r. p)))

En este caso el tamaño de la pila de árboles crece estrictamente en la llamada recursiva, puesto que quitamos un árbol y ponemos dos, pero nótese que la suma del número de nodos de los dos árboles es estrictamente menor que el número de nodos del árbol original porque hemos quitado la raíz. Por tanto, la recursión también está bien definida en este caso.

Por tanto, la pila es vacía en el caso básico y no lo es en el caso recursivo. En este último, según sea vacío o no el árbol en la cima de la pila, los argumentos se transforman de una forma u otra en la llamada recursiva.

Apartado (d)----------------------

Como la recursión es *lineal final* en ambos subcasos, la versión iterativa se obtiene fácilmente. dando lugar al algoritmo siguiente:

fun pre-lista-pila(/ : *lista,* **p :** *pila[árbol-bin])* **dev** *r : lista*
var *a : árbol-bin.* í/ *: pila[árbol-bin]*
r •.= copiar-lista(Z) ; *q* := copiar-pila(p)

mientras -’es-pila-vacía?(</) hacer
a := cima(í/) : desapllar(r/)
si -■es-árbol-vacío?(<i) entonces
añadir-der(r. raiz(<i))
apilar(hijo-dr(a), p)
apilar(hijo-iz(fl), *p)*
fsi
fmientras ffun

Apartado (e)

Ya hemos visto antes que el recorrido en preorden de un árbol binario se obtiene aplicando la operación pre-lista-pila a una lista vacía y una pila que únicamente contiene el árbol a recorrer. Por tanto, el algoritmo iterativo para el recorrido en preorden queda como sigue:

fun preorden-it1 *<J> : árbol-bin)* **dcv** r : *lista*
var <7 : *árbol-bin. q : pila[árbol-bin]*
r Iista-vacía()
q := pila-vacía() ; apilar(¿. *q)*
mientras -■es-pila-vacia?(í/) hacer
a := cima(<;) ; desapilar(g)
si -■es-árbol-vacío?(<í) entonces
añadir-der(;-, raíz(a))
apilar(hijo-dr(«), *p)*
apilar(hijo-iz(<7), *p)*
fsi
fmientras ffun

Suponiendo que las operaciones básicas sobre listas, pilas y árboles que se usan en el algoritmo anterior están implementadas con coste constante, el coste en tiempo de este algoritmo es lineal con respecto al número de nodos del árbol argumento, pues el bucle va

descomponiendo los árboles de la pila (que son los subárboles del árbol de partida) y en cada iteración o bien coloca un nodo en la lista resultado o bien descarta un árbol vacío. Nótese que el número de subárboles vacíos de un árbol binario es comparable al número de nodos del mismo.

El coste en espacio es también lineal con respecto al mismo tamaño.

Finalmente, una posible optimización es asegurarse de no poner árboles vacíos en la pila auxiliar, con lo cual al mismo tiempo que se introducen unos tests desaparece otro pues la pila ya no puede contener árboles vacíos.

fun preorden-it2(/? : *árbol-bin)* **dev** *r : lista*
var *a. h : árbol-bin. q : pila[árbol-bin]*
r := lista-vacía()
si -’es-árbol-vacío?(¿) entonces
q := pila-vacía() : api1ar(¿>, *q)*
mientras -■es-pila-vacía?(g) hacer
a ■.= cirnaGy) ; desapilarfc/)
añadir-der(r, raíz(a))
h := hijo-dr(a)
si -■es-árbol-vacío?(/i) entonces apilar(/i, *p)* fsi
h := hijo-iz(a)
si -’es-árbol-vacío?(/i) entonces apilar(/i. *p)* fsi
fmientras
fsi
6.12.**ffun**

(a) Especificar el *recorrido por niveles* de árboles binarios.

(b) Programar *una función iterativa* para implementar de forma abstracta el recorrido por niveles especificado en el apartado anterior.

(c) Desarrollar una/imcron iterativa niveles-lim tal que niveles-limia. n. *m* (con l < *n* < *m* < alturaía) devuelva el recorrido por niveles de los elementos que se encuentran en el árbol *a* entre los niveles *n* y *ni.* ambos inclusive.

Solución:

Apartado (a)

Enumeramos los niveles de un árbol de forma que su raíz está en el nivel l y que el número del último nivel corresponde a la altura del árbol.

Vamos a especificar el recorrido por niveles de dos formas diferentes.

En la primera consideramos una operación auxiliar que realiza el recorrido de un nivel dado, de forma que para calcular el recorrido por niveles vamos a concatenar los recorridos de todos los niveles, desde l hasta la altura del árbol. La concatenación de los recorridos de varios niveles se realiza mediante otra operación auxiliar que devuelve el recorrido desde la raíz hasta un nivel dado. Estas dos funciones auxiliares son totales, considerando que el recorrido del árbol vacío, o del nivel 0. o de un nivel mayor que la altura es la

lista vacía.

Nótese que se usa la especificación *ÁRBOLES-BINARIOS+[ELEMi* del Ejercicio 6.2 porque allí es donde se definió la operación altura que vamos a necesitaren esta primera versión. especificación *RECORRIDO-NIVELES-ARBOLES-BINARIOS1[ELEM]* usa *NATURALES. ARBOLES-BINARIOS+[ELEM], LISTAS[ELEM]*

operaciones niveles *árbol-bin —> lista*

operaciones privadas

nivel-i-ésimo *árbol-bin nat —> lista*

mveles-hasta-i-esimo*arbol-bin nat —> lista*

variables

a : árbol-bin

i : nat

El recorrido por niveles se define directamente en términos de la segunda operación auxiliar y de la altura. La primera operación auxiliar se define distinguiendo casos sobre el árbol y recursivamente sobre el índice que indica el número de nivel. La segunda operación auxiliar se define recursivamente sobre el índice que indica el número de nivel.

ecuaciones

niveles(o) = niveles-hasta-i-ésimo(<r, altura(a))

nivel-i-ésimo(árbol-vacio. í) = []

nivel-i-ésimo(plantar)*e. dr}.* 0) = []

nivel-i-ésimo(plantar(í;, *e. dr},* l) = [e|

nivel-i-ésimo(plantar(/c. *e, dr}, i)* —

nivel-i-ésimo(í;, í — 1)-H-nivel-i-ésimot</r. í — 1) <= i > 1

niveles-hasta-i-ésimo(«. 0) = []

niveles-hasta-i-ésimo(<í. *i)* = mveles-hasta-i-ésimo(«. *i* — 1) -H- nivel-i-ésimo(u. i) <= í >0 fespecificación

Aunque esta especificación es conecta, si la vemos como algoritmo recursivo para realizar el recorrido por niveles, resulta muy ineficiente pues se repiten los recorridos a lo largo de la estructura arbórea. Para

calcular el recorrido del nivel i-ésimo se pasa por todos los niveles anteriores. Entonces, cuando se hace el recorrido del árbol completo, resulta que por el último nivel se pasa una vez, por el penúltimo nivel dos veces, por el antepenúltimo tres, etc. En particular, cuando tenemos un árbol degenerado (es decir, cuando uno de los hijos de cada nodo es siempre vacío) de *n* nodos, por el nodo raíz se pasa *n* veces, por su hijo *n* — 1. por su “nieto” *n* — 2, hasta llegar a la hoja por la que se pasa una vez; en total, el coste es proporcional a 22?_| *i* 6 ©(n^2).

De cara a obtener el algoritmo iterativo que se pide en el segundo apartado del ejercicio, vamos a ver una segunda especificación de la operación niveles que se basa en usar una estructura auxiliar y que va a dar lugar a un algoritmo más eficiente.

Para recorrer por niveles un árbol, empezamos por la raíz (nivel 1) y debemos continuar con las raíces de todos sus hijos (nivel 2) antes de pasar a los nodos del nivel 3. Como a partir del nivel 2 tenemos que ir tratando todos los hijos, y después de tratar un árbol no podemos pasar al siguiente olvidando el anterior, pues más tarde habrá que recorrer los nodos en el siguiente nivel, necesitamos una estructura donde se vayan guardando los árboles pendientes.

Después de recorrer la raíz del primer árbol pendiente hay que pasar a la raíz del segundo árbol, pero acordándose de los niveles (o sea. árboles) pendientes del primer árbol. Ahora bien, no hay que pasar a tratar estos árboles hasta haber terminado de tratar los pendientes que hay en ese momento, por lo que esos árboles se guardan añadiéndolos al final de los pendientes de tratar en la estructura.

De esta forma, la estructura adecuada es una *cola de árboles binarios* de tipo *cola[árbol-bin],* puesto que las operaciones que necesitamos son añadir por un extremo y eliminar por el otro.

Cuando en la cola de árboles pendientes encontramos un árbol vacío, simplemente lo descartamos, porque no tiene ningún nodo para recorrer.

especificación *RECORRIDO-NIVELES-ÁRBOLES-BINARIOS2[ELEMI*

usa *ÁRBOLES-BINARIOS[ELEM], COLAS[ÁRBOLES-BINARIOS[ELEM]J. LISTAS[ELEM]*

operaciones

niveles : *árbol-bin —* lista*

operaciones privadas

niveles-cola : *cola[árbol-bini —> lista*

variables

a : árbol-bin

c : cola[árbol-bin]

Para definir el recorrido por niveles de un árbol binario, basta considerar una

cola unitaria con ese árbol y llamar a la operación auxiliar que va haciendo el recorrido por niveles de todos los árboles en la cola. Nótese que *no* se recorre por niveles cada árbol por separado para después concatenar los resultados, sino que se recorren los primeros niveles de todos los árboles, después los segundos niveles de todos los árboles, etc.

ecuaciones

niveles(n) = niveles-cola(pedir-vez(cola-vacía, *a))*

Cuando la cola es vacía, el recorrido es la lista vacía. Si la cola no es vacía, pero el primer árbol es vacío, se descarta y se continúa. Finalmente, si la cola no es vacía y el primer árbol tampoco, este se descompone y se procede recursivamente como se ha explicado anteriormente, quitando el primer árbol de la cola pues ya ha sido tratado y añadiendo su hijo izquierdo y luegosu hijo derecho.

niveles-cola(c) = [] <= es-cola-vacía?(c)

niveles-cola(c) = niveles-cola(avanzar(c))
<= ~'es-cola-vacía?(c) A es-árbol-vacío?(primero(c))

niveles-cola(c) = raiXprimerofc)): niveles-cola(pedlr-vez(pedir-vez(avanzar(c),
híjo-iz(primero(c))).
hijo-dr(primero(c))))

fespecificación <= -■es-cola-vacía?(c) A -

■es-árbol-vacío?(primero(c))

En la parte derecha de la tercera ecuación el tamaño de la cola de árboles crece estrictamente en la llamada recursiva, puesto que quitamos un árbol y ponemos dos. pero nótese que la suma del número de nodos de los dos árboles es estrictamente menor que el número de nodos del árbol original porque hemos quitado la raíz. Por tanto, la operación está bien definida.

Apartado (b)-----------------------

Pasamos a iterativo el algoritmo recursivo correspondiente a las ecuaciones del apartado anterior, y lo optimizamos no añadiendo los árboles vacíos a la cola auxiliar. De esta forma, no hace falta considerar el caso de un árbol vacío (que simplemente se descana) porque nunca aparece en la cola.

fun niveles(<? : *árbol-bin)* **dev / :** *lista*
var *sig. hijo : árbol-bin, c : cola[árbol-bin]*
l := lista-vacíaO
si -■es-árbol-vacío?*(a)* entonces
c cola-vacíaO ; pedir-vez(c. *a)*
mientras -■es-cola-vacía?(c) hacer
sig := primero(c) ; avanzar(c)
añadir-der(/. raíz(.ríg)) *hijo* .= hijo-iz(síg)
si -■es-árbol-vacío? (/it/o) entonces pedir-vez(c, *hijo)* fsi
hijo := hijo-dr(.sz^)
si -■es-árbol-vacío?(/ií/'o) **entonces** pedir-vez(c. *hijo)* **fsi fmientras**

fsi
ffun

Si se compara este algoritmo iterativo para el recorrido por niveles de un árbol binario con el algoritmo iterativo preorden-it2 para el recorrido en preorden obtenido al final del Ejercicio 6.11. se podrá observar que siguen exactamente la misma idea, con la diferencia fundamental de que aquel usa una pila, mientras que este usa una cola.

Suponiendo que las operaciones básicas sobre listas, colas y árboles que se usan en el algoritmo anterior están implementadas con coste constante, el coste en tiempo de este algoritmo es lineal con respecto al número de nodos del árbol argumento, pues el bucle va descomponiendo los árboles de la cola (que son los subárboles del árbol de partida) y en cada iteración coloca un nodo en la lista resultado. El coste en espacio es también lineal con respecto al mismo tamaño.

Apartado (c)-------------------

Utilizamos la misma idea que en los apartados anteriores, pero guardamos

tipos
árbol-nivel
ftipos

cada árbol en la cola junto con información sobre el nivel en que se encuentra su raíz, siempre con respecto al árbol original. Cuando se trata un árbol, solamente se añade su raíz al recorrido

si su información asociada indica que está en el rango apropiado. En la cola solamente hay árboles cuya raíz está en los niveles entre l y *in* en el árbol original. Para ello, antes de añadir un hijo a la cola se comprueba su nivel.

reg
árbol : árbol-bin
nivel : nat+
freg

{ 1 < n < m < altura(n) }

fun niveles-lim(« : *árbol-bin. n. ni* : naf1-) dev / : *lista* **var** *sig : árbol-bin, p : árbol-nivel, c : cola[árbol-nivel]*

l := lista-vacíaQ
p.árbol := a ; p.nivel := I
c := cola-vacía() ; pedir-vez(c, *p)*
mientras -'es-cola-vacia? (c) hacer *p* := primero(c) ; avanzar(c) sig := *p.árbol; k := p.nivel* **si** *n < k A k < ni* **entonces** añadir-der(i, raíz(síg)) **fsi**
si *k* < *ni* **entonces**
p.nivel := k + 1 *p.árbol* := hijo-iz(síg) si -'es-árbol-vacío?(p.árZio/) entonces pedir-vez(c. *p)* fsi *p.árbol* hijo-dr(síg)
si -•es-árbol-vacio?(*p.árbol)*
entonces pedir-vez(c, *p)* **fsi fsi**
fmientras
ffun

Nótese que a diferencia del algoritmo del apartado anterior, en este no consideramos el caso vacío para el árbol *a* porque en las condiciones del enunciado ese caso no puede darse.

El coste en tiempo y espacio es lineal

con respecto al número de nodos del árbol argumento

6.13. Suponemos disponible un tipo *personaje* con dos operaciones es-dragón? y es-princesa? que permiten averiguar si un personaje es un dragón o una princesa, respectivamente. Suponemos también que ningún personaje es dragón y princesa a la vez, pero que un personaje puede no ser ninguna de las dos cosas.

Dado un árbol binario del tipo *árbol-bin[personaje],* se denominan *nodos accesibles* aquellos nodos tales que a lo largo del camino desde la raíz hasta el nodo (ambos inclusive) no se encuentra ningún dragón.

(a) Especificar una operación que calcule el número de nodos accesibles de un árbol *a* ocupados por una princesa.

(b) Programar una función iterativa de coste lineal que encuentre una princesa accesible *lo más cerca posible de la raíz* de un árbol dado *a,* suponiendo que algún nodo accesible de *a* contiene una princesa.

----------------Solución------------------

Apartado (a)

Las suposiciones del enunciado adquieren la forma de parámetro en la especificación.

parámetro *PERSONAJES*
usa *BOOLEANOS* **tipos** *personaje*
operaciones
es-dragón? : personaje → bool
es-princesa?: personaje → bool
variables

.v : *personaje*
ecuaciones
es-dragón?(.r) = falso <= es-princesa?(x)
es-princesa?(,v) = falso <= es-dragón?(x)
fparámetro

La especificación proporciona una operación que calcula el número de princesas accesibles en un árbol de personajes.

especificación *PRINCESAS-Y-DRAGONES[PERSONAJES]* **usa** *ÁRBOLES-BINARIOSjPERSONAJESj, NATURALES* **operaciones**
núm-princesas-acc : *árbol-bin[personajel* —> nat **variables**
A' *: personaje*
iz. dr : arbmjpersonaje]

La operación núm-princesas-acc se define dislinguiendo en principio casos sobre las constructoras de árboles binarios. Si el árbol es vacío, el número de princesas accesibles es cero. Cuando el árbol no es vacío, distinguimos subcasos según el tipo de personaje en la raíz. Si ese personaje es un dragón, todos los nodos del árbol son no accesibles y. por tanto, en particular el número de princesas accesibles es cero. Si no es un dragón, el número de princesas accesibles es la suma de las princesas accesibles del hijo izquierdo y del hijo derecho, más uno si el personaje de la raíz es una princesa.

ecuaciones

núm-princesas-acc(árbol-vacío) = 0
núm-príncesas-acc(plantar(iz. A', r/r))
= 0 e= es-dragón?(x)
núm-princesas-acc(plantar(íc. A, *dr))* =
núm-princesas-acc(íc) + núm-princesas-acc(</r) + l
<= es-princesa?(x)
núm-princesas-acc(plantar(íz, x, *dr))* =
núm-princesas-acc(í;) + núm-princesas-acc(dr)
<= -■es-dragón?(x) A -■es-princesa?(x) **fespecificación**

Apartado (b)---------------------

Para encontrar una princesa accesible lo más cerca posible de la raíz hay que recorrer el árbol por *niveles,* desechando aquellos subárboles cuya raíz sea un dragón. El recorrido se puede detener cuando se encuentre la primera princesa, que existe por la hipótesis del enunciado.

Por tanto, podemos adaptar el algoritmo iterativo niveles en la solución del Apartado (b) del Ejercicio 6.12, que realiza un recorrido por niveles de un árbol binario utilizando como estructura auxiliar una cola de árboles binarios. Las adaptaciones consisten en quitar el caso vacío que no puede darse, controlar el bucle con una variable booleana que permite terminarlo tan pronto como se encuentre una princesa accesible en vez de esperar a que la cola sea vacía, tratar el personaje de salida en vez de acumular el recorrido en una lista, y

distinguir casos sobre el personaje para terminar cuando se trate de una princesa, continuar con el recorrido de la forma habitual cuando no sea princesa ni dragón, y continuar (de forma implícita) pero sin poner los hijos en la cola cuando se trate de un dragón.

(núm-princesas-accfn) > 0)

fun prtncesa(<i : *árbol-bin[personaie]***)**
dev x *: personaje*
var *sig, hijo : árbol-binjpersonaje], c : cola[árbol-bin[personajejj*
c := cola-vacia() ; pedir-vez(c, *a)*
encontrada := falso
mientras —•*encontrada* **hacer**
sig := primero(c) ; avanzar(c)

x := raiz(síg)
si es-princesa? (x) **entonces**
encontrada := cierto
si no
si -■es-dragón?(x) entonces
hijo := hijo-iz(sig)
si -’es-árbol-vacío(7ii/o) **entonces**
pedir-vez(c. *hijo)* fsi
hijo := hijo-der(síg)
si -’es-árbol-vacío(/i(/o) **entonces**
pedir-vez(c. *hijo)* **fsi**
fsi
fsi
fniientras
ffun

En el caso peor, el recorrido tiene que llegar al final para encontrar la princesa accesible, por lo que el coste en tiempo de este algoritmo es lineal con respecto al número de nodos en el árbol de personajes.

Especificar un TAD para describir los *árboles generales* cuyos nodos contienen elementos pertenecientes a un tipo dado como parámetro, y al menos las siguientes operaciones:

- construir un árbol a partir de un elemento y una sucesión de árboles del mismo tipo.
- construir sucesiones de árboles generales, incluyendo la sucesión vacía.
- consultar la raíz.
- calcular la sucesión de hijos.
- calcular el número de hijos,
- calcular el hijo í-ésimo, y
- determinar si un árbol es una hoja.

---------------Solución----------------

Al igual que el TAD de los árboles binarios, el TAD de los árboles generales está parametrizado por el parámetro más sencillo *ELEM.*

Como el número de hijos de un nodo es variable, necesitamos un tipo para representar sucesiones de árboles, que se comportan como listas de árboles. Como no podemos instanciar un tipo paramétrico (listas en este caso) con un tipo que está siendo definido al mismo tiempo y como tampoco necesitamos todas las operaciones sobre listas, vamos a definir simultáneamente un tipo para sucesiones de árboles que denominamos "bosques".

En varias ocasiones, una operación definida sobre árboles va ti ir acompañada de una operación semejante

sobre bosques, por lo que en la siguiente especificación, además de las operaciones del enunciado, aparecen varias operaciones de ese carácter.

especificación *ÁRBOLES-GENERALES[ELEM]* **usa** *BOOLEANOS. NATURALES* **tipos** *árbol, bosque*
operaciones
plantar : *elemento bosque* —> *árbol* (constructora)
bosque-vacío : —> *bosque* (constructora)
añ-árbol : *árbol bosque* —> *bosque* (constructora)
raíz : *árbol* —> *elemento*
hijos : *árbol* —> *bosque*
núm-hijos *árbol* —> *nat*
longitud : *bosque* —> *nat*
subárbol : *árbol nat* —> $_p$ *árbol*
_[_] : *bosque nat* —> $_p$ *árbol*
es-hoja? *árbol* —> *bool*
variables
e : elemento
a : árbol
b : bosque
í *: nat*

Las operaciones bosque-vacío y añ-árbol son las constructoras libres del **tipo** *bosque* **que. como** ya hemos comentado, consiste en listas de árboles, aunque no utilicemos ese nombre.

La operación plantar es la única constructora, también libre, del tipo *árbol.*

Nótese que los tipos *bosque* y *árbol* se definen de forma mutuamente recursiva. Un árbol se construye a partir de un bosque y a su vez un bosque consta de

una lista de árboles. El caso básico se obtiene cuando el bosque es vacío, dando lugar a un árbol que no tiene hijos, es decir, una hoja. Así pues, en el tipo de los árboles generales *no* hay árbol vacío.

Las operaciones raíz e hijos son las ''destructoras" o proyecciones asociadas a la constructora plantar, por lo que sus ecuaciones son inmediatas.

ecuaciones

raiz(plantar(e. *b)) = e*

hijos(plantar(<-, Z?)) = *b*

El número de hijos de un árbol es la longitud de la lista formada por sus hijos.

num-hijos(plantar(e./?)) = longitud(Zz)

longitud(bosque-vacío) = 0

longitud!añ-árbol(rr./?)) = 1 + longitud(b)

El subárbol (o hijo) i-ésimo de un árbol dado *a* es el árbol que ocupa la posición *i* en su bosque de hijos, por lo que es una operación *parcial* que solo tiene sentido cuando el índice *i* está en el rango entre I y núm-hijos(íz). Se define en términos de una operación equivalente sobre bosques, cuya versión sobre listas aparece en el Ejercicio 5.8. Esta operación se define por recursión sobre el índice.

subárbol(plantar(e./?), í) = ¿>[r]

b[i) = error <= *i* == 0 v *i* > longitud(Zz)

añ-árbol(rr. b)[11 = *a*

añ-árbol(<r. ¿>)[/] = ¿>[z — 1] 4= I <*i A i* < longitud(b) + 1

El predicado es-hoja? es fácil de definir

puesto que un árbol es una hoja cuando su bosque de hijos es vacío. Una forma de caracterizar esta situación consiste en comprobar que el número de hijos, que coincide con la longitud del bosque de hijos, es cero. Otra posibilidad que no detallamos sería distinguir casos sobre las constructoras.

es-hoja?(<¿) = (núm-hijos(rr) == 0)

fespecificación

Extender la especificación de los árboles generales del Ejercicio 6.14 con las operaciones siguientes:

- **calcular la altura de un árbol,**
- **calcular el número de nodos.**
- **calcular el número de hojas, y**
- **calcular el grado de un árbol.**

Solución:

Las cuatro operaciones sobre árboles se definen mediante recursión mutua con operaciones apropiadas sobre bosques. Por ejemplo, la altura de un árbol se calcula teniendo en cuenta una operación adecuada de "altura" sobre bosques. Estas operaciones sobre bosques aparecen como operaciones privadas en la especificación.

especificación *ÁRBOLES-GENERALES+[ELEM]*

usa *ÁRBOLES-GENERALES[ELEM], NATURALES*

operaciones

altura*: árbol—>* *nat*

núm-nodos : *árbol —»* *nat*

núm-hojas *árbol —>* nat

grado*árbol —>* *nat*

operaciones privadas

altura-bosque *: bosque —» nat*

núm-nodos-bosque : *bosque —> nat*

núm-hojas-bosque: *bosque —> nat*

grado-bosque *: bosque —> nat*

variables

e : elemento

a : árbol

b : bosque

La altura de un árbol es uno más (por la raíz) que el máximo de todas las alturas de sus hijos. Esta idea se concreta mediante una operación de "altura" sobre bosques que calcula el máximo de todas las alturas de los árboles que forman el bosque. Así, la operación sobre árboles es mutuamente recursiva con la operación correspondiente sobre bosques.

altura(plantar(e, ó)) = 1 + altura-bosque(b)

altura-bosque(bosque-vacío) = 0

altura-bosque(añ-árbol(a,/>)) = máx(altura(a). altura-bosque(b))

El número de nodos de un árbol es uno más (por la raíz) que el número total de nodos en todos sus hijos. Así pues, definimos una operación sobre bosques que calcula la suma de los números de nodos en todos los árboles que forman el bosque. De nuevo, ambas operaciones son mutuamente recursivas.

núm-nodos(plantar(e, *b))* = 1 + núm-nodos-bosque(ít)

núm-nodos-bosque (bosque-vacío) = 0

núm-nodos-bosque(añ-árbol(«, *b))* = núm-nodos(<7) + núm-nodos-bosque(ó)

Si un árbol no tiene hijos, es una hoja y por tanto el número de hojas es uno. Si tiene hijos, no es hoja y el número de hojas del árbol original se obtiene al

considerar el número total de hojas en los hijos, lo cual se define mediante una operación sobre bosques.

núm-hojas(plantar(e. bosque-vacío))
= 1

núm-hojas(plantar(e. añ-árbol(a, Z>)))
= núm-hojas-bosque(añ-árbol(«. *b))*

núm-hojas-bosque(bosque-vacío) = 0

núm-hojas-bosque(añ-árbol(<7, ¿)) = núm-hojas(u) 4-núm-hojas-bosqueiói

El grado de un árbol es el número máximo de hijos de todos los nodos que forman el árbol. Así pues, consideramos una operación sobre bosques que calcula el máximo de todos los grados de los árboles en un bosque.

grado(plantar(e. /?)) = máx(longitud(¿), grado-bosque(i))

grado-bosqueíbosque-vacío) = 0

grado-bosque(añ-árbol(rt./?)) = máx(grado(u). grado-bosque(b))

fespecificación

6.16. Diseñar una representación dinámica del TAD de los árboles generales y desarrollar una implementación de las operaciones especificadas en el Ejercicio 6.14.

------------------Solución-----------------

Una primera posibilidad es representar un nodo de un árbol general como un par formado por un elemento y una lista de árboles de la misma clase (es decir, un bosque), la cual a su vez se representa usando cualquier representación dinámica de las listas, o incluso usando las listas de forma abstracta, sin acceso a su representación concreta.

Otra posibilidad es representar un nodo de un árbol general como un registro con *tres* campos: un elemento. un enlace al primer hijo (el "primogénito"), y un enlace al árbol siguiente, el "siguiente hermano" en la lista o bosque de hijos. La principal ventaja de esta representación es que se utiliza el mismo tipo de nodos para árboles y para bosques que. como sabemos, se comportan como listas de árboles. Así. un árbol general se representa como un enlace a su nodo raíz, y un bosque de tales árboles se representa como un enlace al nodo raíz del primer árbol en la lista si no es vacía o como el enlace nulo en caso contrario. Detallamos a continuación varios ejemplos:

- Cuando tenemos un único árbol que es una hoja, como no tiene hijos ni hermanos, está representado por un enlace a un nodo en el que ambos enlaces, primogénito y siguiente hermano, son nulos.
- Cuando tenemos un único árbol que no es una hoja, como no tiene hermanos pero sí hijos, está representado por un enlace a un nodo en el que el enlace primogénito apunta al primer hijo y el enlace siguiente hermano es nulo.
- Un bosque vacío se representa con un enlace nulo.
- Un bosque formado por un único árbol se identifica con ese árbol.
- Un bosque formado por más de un árbol se representa por un enlace al nodo raíz del primer árbol, el cual tiene hermanos, por lo que en ese nodo el enlace siguiente hermano no es nulo y apuntará al segundo árbol en el bosque.

En definitiva, los tipos representantes son los siguientes:

tipos
enlace-árbol = **puntero a** *nodo-árbol*
nodo-árbol **= reg**
valor : elemento
príniog, sig-herin : enlace-árbol
freg
árbol = *enlace-árbol*
bosque = *enlace-árbol*
ftipos

Nótese que salvo el cambio en los nombres de los campos, la estructura es equivalente a la utilizada en el Ejercicio 6.3 para representar los árboles binarios (por esa razón hemos mantenido el mismo nombre en los tipos también). De hecho, esta idea justifica una forma sistemática de representar cualquier árbol general como árbol binario.

fun plantarte : *elemento, b : bosque)* **dev** *a : árbol* { 0(1))
reservar(o)
a T *.valor := e*
a f *.prímog := b*
a f *.sig-henn* := nil
ffun
fun bosque-vacíoO **dev** *b : bosque* (0(1) }
b := nil **ffun**
proc añ-árbol(« : *árbol, b : bosque)* { 0(1))
a f *.sig-henn := b*
b := a
fproc
fun raíz(a : *árbol)* dev *e : elemento* (0(1) }
e := a f *.valor* **ffun**
fun hijos(rí : *árbol)* **dev** *b : bosque* (0(1) }
b := a t *.prímog*
ffun
fun núm-hijos(« : *árbol)* **dev** *n : nat*
n := longitud(« f *.prímog)*
ffun
fun longitud(6 *; bosque)* dev *n : nat*

```
   var c : enlace-árbol
    c := b ; n := 0
    mientras c / nil hacer
      n := n + I
      c := c I .sig-henn
    fmientras
   ffun
   fun subárbol (o : árbol, i : nat) dev h
: árbol ( 0(í) }
    h := consultaría t .prímog. i)
   ffun
   fun consultará : bosque, i : nat) dev h
: árbol ( 0(í) )
    si i = 0 entonces error(índice no
válido)
    si no
      h := b ; j := 1
      mientras h / nil A j i hacer
        j := j + 1
        h := h f .sig-henn
      fmientras
      si h = nil entonces error( índice no
      válido) fsi
    fsi
ffun
```

fun es-hoja?(rr : *árbol)* **dev** *r : bool ((-)(!))*

r (a f *.primog* = nil) **ffun**

El coste de todos los algoritmos es constante, excepto longitud y núm-hijos que son lineales con respecto al número de árboles en el bosque, y consultar y subárbol que son lineales con respecto al valor del segundo argumento.

Especificar e implementar de forma abstracta operaciones sobre árboles generales para calcular

- el recorrido en *preorden,*
- el recorrido en *postorden.*
- el recorrido por *niveles,* y
- *\a frontera.* es decir, la lista formada por los elementos almacenados en las hojas del árbol, tomados de izquierda a derecha.

-------------------Solución----------------

Como ya ocurría en la solución del Ejercicio 6.15. las operaciones sobre árboles se definen con la ayuda de operaciones auxiliares sobre bosques que son operaciones privadas en la siguiente especificación.

especificación *RECOFtFIIE>OS-ÁFiBOLES-GENERALES[ELEM]* **usa** *ARBOLES-GENERALES[ELEM], LISTAS[ELEM]* **operaciones**

preorden*árbol—> lista*

postorden *árbol —> lista*

niveles*árbol —> lista*

frontera*árbol —» lista*

operaciones privadas

preorden-bosque *bosque —> lista*

postorden-bosque *bosque* —> *lista*

niveles-bosque *bosque* —» *lista*

_ -H- _*bosquebosque* —> *bosque*

frontera-bosque *bosque* —> *lista*

variables *e : elemento a . árbol b. b' : bosque*

El recorrido en preorden visita primero la raíz del árbol y después recorre en preorden todos sus hijos, para lo cual usamos una operación sobre bosques que concatena los recorridos en preorden de todos los árboles que forman un bosque. La misma idea se aplica al recorrido en postorden, con la diferencia de que la raíz se visita al final del recorrido.

preorden(plantar(c. *b))* = [<?] -H- preorden-bosque(¿)

preorden-bosque (bosque-vacío) = []

preorden-bosque(añ-árbol(rr, *b))* = preorden(n) -++ preorden-bosque(Z>)

postorden(plantar(e, ¿>)) = postorden-bosque(/>)-H-[e|

postorden-bosque(bosque-vacío) = {1

postorden-bosque(añ-árbol(<¿. />)) = postorden(n) -H- postorden-bosque(/ri

El recorrido por niveles empieza por la raíz del árbol en el nivel 1 y pasa a recorrer todas las raíces de los hijos en el nivel 2. Para ello, se usa una operación auxiliar que va haciendo el recorrido por niveles de todos los árboles en un bosque. A diferencia de los dos recorridos anteriores, la operación auxiliar *no* concatena los recorridos por niveles de cada árbol en el bosque por separado, sino que recorre los primeros niveles de todos los árboles, después los segundos niveles de todos los árboles, etc.

Al tener disponible el tipo *bosque* no necesitamos usar en la especificación colas de árboles, como en la solución del Ejercicio 6.12. Finalmente, para poner los árboles pendientes al final de los que se van a tratar a continuación, usamos una operación auxiliar sobre bosques que corresponde a la concatenación de listas, y para la cual vamos a utilizar la misma notación.

En resumen, la ecuación fundamental en la definición del recorrido por niveles es la tercera de las siguientes, en la cual es necesario tener en cuenta las componentes del primer árbol del bosque (nótese que no hay más que una posibilidad porque no hay árbol vacío) y es esencial el orden en que se concatenan los bosques *b'* y *b* en la llamada recursiva en la parte derecha de la ecuación.

niveles(plantar(e. b)) = [<?] -H- niveles-bosque(b)

niveles-bosque(bosque-vacío) = []
niveles-bosque(añ-árbol(plantar(e. b'), b)) = [e] -H- mveles-bosque(b 4+ b')
bosque-vacío -H- b = b
añ-árbol(«. b') -H- b = añ-árbol(n, *b′* -H- b)

Como la frontera va recorriendo las hojas de un árbol, la distinción de casos es la misma que en la especificación de la operación para calcular el número de hojas en la solución del Ejercicio 6.15. incluyendo una operación auxiliar frontera-bosque que concatena las fronteras de todos los árboles en un bosque.

frontera(plantar(e, bosque-vacío)) = [e]
frontera(plantar(e, añ-árbol(«, b))) = frontera-bosque(añ-árbol(n. b))
frontera-bosque(bosque-vacío) = [ ]
frontera-bosque(añ-árbol(a. b)) = frontera(rr) -H- frontera-bosque(b)

fespecificación

Consideremos ahora la implementación abstracta de cada una de las operaciones. Para los recorridos en preorden y postorden, y la frontera, la idea más sencilla es considerar una implementación recursiva con un bucle que va haciendo llamadas sobre cada hijo. En los bucles correspondientes a los dos primeros, cuando el árbol es una hoja no se hace nada, debido a que el rango del bucle es vacío. En el tercer algoritmo. el caso de una hoja hay que tratarlo por separado, pues la frontera solamente recoge la información en las hojas.

fun preordenfa : *árbol)* **dev** *I : lista*
var b : *bosque, k : lista*
I := unitaria(raíz(a))
b := hijos(a) ; *n* := longitud(b)
para *i* = 1 **hasta** *n* **hacer**
k preorden(consultar(b, í))
/ := concatenare/,/.')
fpara
ffun
fun postorden(« : *árbol)* **dev / :** *lista*
var *b : bosque, k : lista*
/ Iista-vacía()
b := hijos(a) ; *n* := longitud(b)
para *i* = I **hasta** *n* **hacer**
k := postorden (consultar!/?, /))
/ := concatenar//, *k)*
fpara
añadir-der(/. raíz!»))
ffun
fun frontera(o : *árbol)* **dev / :** *lista*
var /? : *bosque, k : lista*
si es-hoja?(«) **entonces / :=** unitaria(raíz(«))
si no
/ := lista-vacía!)
/? .= hijos(o) ; *n* := longitud!/?)
para *i* = l **hasta** *n* **hacer**
k := frontera(consultar(/?,/))
/ := concatenar^, *k)*
fpara
fsi
ffun

Para el recorrido por niveles, consideramos una implementación iterativa que usa una cola de árboles, semejante a la vista en la solución del

Apartado (b) del Ejercicio 6.12. La principal diferencia es que en vez de poner dos árboles en la cola tenemos que poner todos los hijos en cada caso, para lo cual tenemos un bucle que recorre el bosque.

fun niveles!» : *árbol)* **dev /** *: lista*
var *sig, hijo : árbol, /? : bosque, c : colajárbol]*
  si es-hoja?(») **entonces / :=** unitaria(raiz(«))
  si no
    / = lista-vaciaO
    c := cola-vacia!) ; pedir-vez(c, *a)*
    mientras -■es-cola-vacía?(c) hacer
      sig := primero(c) ; avanzar(c)
      añadir-der(/, raíz(szq))
      /? := hijosf.vrj?) ; *u* := longitudfZ?)
      para *i* = **I hasta** *n* **hacer**
        hijo := consultará, *i)*
        pedir-vez(c. *hijo)*
      fpara
    fmientras
  fsi
ffun

Otra posibilidad sería utilizar el mismo bosque en vez de una cola de árboles, de forma semejante a la especificación de la operación más arriba, pero para esto necesitamos tener implementadas más operaciones sobre bosques, como concatenar dos bosques, consultar y eliminar el primer árbol en un bosque, etc.

Como la implementación de la operación consultar que se ha hecho

en el Ejercicio 6.16 es lineal con respecto al valor de la posición, el coste de estas implementaciones abstractas de los recorridos es cuadrático con respecto al número de nodos del árbol que se recorre. Este coste se puede mejorar realizando una implementación sobre la representación dinámica, perdiendo a cambio de la mayor eficiencia la abstracción.

Capitulo 7

7. ÁRBOLES BINARIOS DE BÚSQUEDA Y TABLAS

INTRO

Una de las clases de libros más útiles en diferentes facetas de la vida cotidiana está constituida por los *diccionarios:* de una lengua concreta, bilingües, temáticos, de sinónimos, de dudas, etc. También otros libros como la guia telefónica pueden incluirse en esa misma clase. La estructura de *tabla* como listado de información asociada a claves, tal vez. sin ordenar, es mucho más general y aparece por doquier, como demuestra por poner un ejemplo, la popularidad de las hojas de cálculo. La idea adicional que aparece en la noción de diccionario es que la información está *ordenada según la clave,* lo cual es fundamental para poder buscar información de forma sencilla y eficiente.

Teniendo en cuenta que los usuarios de los diccionarios no solamente son los lectores sino también los autores de los mismos, las operaciones más importantes sobre estas estructuras son buscar, insertar y borrar. Obviamente, para que al realizar una búsqueda el resultado no se preste a confusión, es esencial que

cada clave aparezca solamente una vez. es decir, no se permiten repeticiones en las claves, tanto en tablas como en diccionarios.

Aunque también consideramos la idea de que se guarde un par formado por una clave y la información asociada, para facilitar la presentación y comprensión de las propiedades de estas estructuras, en principio consideramos que la información se identifica con la clave, dando lugar a elementos con un orden total estricto, ya que no puede haber elementos repetidos.

En este capítulo tratamos ambas estructuras, tablas y diccionarios, y dedicamos especial atención a la implementación de diccionarios en términos de *árboles binarios de búsqueda,* que son árboles binarios cuyos nodos guardan elementos sobre los cuales hay definido un orden total estricto y que satisfacen la siguiente propiedad adicional: el elemento en cada nodo es mayor que todos sus descendientes izquierdos y menor que todos sus descendientes derechos. Equivalentemente, o bien el árbol es vacío, o bien el elemento en la raíz es mayor que los elementos del hijo izquierdo y menor que los elementos del hijo derecho, y recursivamente los dos hijos son a su vez árboles binarios de búsqueda.

Como veremos en el Ejercicio 7.2. la eficiencia de las operaciones sobre árboles de búsqueda depende de la altura

del árbol. Aunque no vamos a tratarlas en este libro por considerarlas demasiado avanzadas y bien tratadas en varios libros de texto, es importante recordar que existen diferentes implementaciones de los árboles de búsqueda para asegurar la eficiencia de las operaciones: *árboles* .41*L. rojinegros.* 2-3, etc. Para mayor información sobre ellas, remitimos al lector a libros sobre estructuras de datos, incluyendo ICLRSOI, Fra94, HS94, WeiOOJ.

EJERCICIOS RESUELTOS

7.1. Especificar un TAD para describir los árboles binarios de búsqueda con elementos pertenecientes a un tipo dado como parámetro, con las siguientes operaciones:

- crear un árbol binario de búsqueda vacío,
- insertar un elemento,
- determinar si un elemento pertenece al árbol,
- consultar el menor elemento,
- consultar el mayor elemento,
- eliminar un elemento, y
- determinar si el árbol es vacío.

-----------------**Solución**----------------

Los elementos almacenados en un árbol de búsqueda tienen que tener definida una operación de igualdad para poder, por ejemplo, determinar si un elemento pertenece al árbol. Es esencial que también tengan definida una relación de orden total estricta (_ < _), ya que este orden influye en la estructuración de los elementos en el árbol. El siguiente parámetro, que extiende el parámetro *ELEM=* definido en la Sección 1.1.5, define las propiedades que estas operaciones deben satisfacer.

parámetro *ELEM=<*
usa *BOOLEANOS. ELEM=*
operaciones
_ < _ *: elemento elemento —> bool*
_ > _ *: elemento elemento —> bool*
variables
x. y, z : *elemento* **ecuaciones**

x < x = falso(antirreflexividad
i
x < z = cierto <= x < *y* A *y* < z
{ transitividad)
x < yv y < x = cierto <= x /
y(totalidad)
x > y = y < x

fparámetro

Una primera posibilidad para especificar los árboles binarios de búsqueda es realizar una extensión de los árboles binarios con las nuevas operaciones, ya que los primeros son un subconjunto de los segundos. Pero así no tendríamos una separación clara entre los dos tipos y las nuevas operaciones serían parciales, al no estar definidas para árboles que no fueran de búsqueda.

Otra posibilidad, que aquí seguimos, consiste en especificar los árboles binarios de búsqueda partiendo de cero (aunque la relación con los árboles binarios esté siempre implícitamente presente). Como constructoras necesitamos una que cree el árbol vacío (abb-vacío) y otra que nos permita construir árboles con más elementos. Si tomáramos insertar como constructora, perderíamos la estructura jerárquica que tienen los árboles basada en la raíz, el hijo izquierdo y el derecho. Por tanto, utilizaremos como constructora una operación plantar que, a partir de dos árboles de búsqueda *y* un elemento, construye un nuevo árbol de búsqueda que tiene como raíz el elemento dado y

como hijos izquierdo y derecho, los árboles dados. Dicha constructora es *parcial* ya que solo está definida cuando el elemento dado como raíz es mayor que los elementos del hijo izquierdo y menor que los elementos del hijo derecho; es decir, el árbol construido sigue siendo de búsqueda. Sin embargo, aunque plantar sea una de las constructoras, cuando se utilicen los árboles de búsqueda generalmente se construirán a partir del árbol vacío insertando elementos. mediante la operación insertar. Las restantes operaciones se definen de forma natural en la siguiente signatura:

especificación *ÁRBOLES-BINARIOS-DE-BÚSOUEDA[ELEM= <]* **usa** *BOOLEANOS*

tipos *árbol-bb*

operaciones abb-			
vacío plantar		*árbol-bb*	{ constructora
insertar está?	*árbol-bb elemento árbol-*	*árbol-bb*	) {
mínimo	*bb elemento árbol-bb*	*árbol-bb*	
máximo	*elemento árbol-bb árbol-*	*bool*	
eliminar es-	*bb*	*elemento*	
abb-vacío?	*árbol-bb*	*elemento*	
variables	*elemento árbol-bb árbol-*	*árbol-bb*	
	bb	*bool*	

e. f : elemento iz. dr : árbol-bb

Las constructoras son *libres,* pero plantar es *parcial,* por lo que damos una ecuación que expresa cuándo no está definida la operación, según la definición de árbol binario de búsqueda (Sección 7.1). Para ello se utilizan las operaciones mínimo y máximo, que devuelven el elemento mínimo y máximo de un árbol no vacío, respectivamente. El árbol plantar(í;, *e. dr}* es correcto si *e* es mayor que todos los elementos en *iz* (es decir, *e* > máximo!/;)) y menor que todos los elementos en *dr* (es decir, *e* < mínimo(í/r)). Nótese el uso del orden estricto para evitar repeticiones.

ecuaciones
plantar!/;, *e. dr)*

error <= -’(es-abb-vacío?(/;) v *e* > máximo!/';)) v -’(es-abb-vacío?(r/r) v *e* < mínimo(Jr))

A partir de ahora supondremos que todos los términos construidos con plantar son correctos, es decir, representan árboles binarios de búsqueda.

Insertar un elemento en un árbol vacío construye un árbol que tiene como raíz al elemento, y dos hijos vacíos. En otro caso, si el elemento está ya en la raíz, el árbol no se modifica: si el nuevo elemento es menor que la raíz, entonces se inserta de forma recursiva en el hijo izquierdo; y si es mayor que la raíz, se inserta recursivamente en el hijo derecho.

insertarle, abb-vacío)
insertarle, plantar!/;, e. *dr})*
insertarle, plantar!*iz. f. dr))*
insertarle, plantar!/;, *f. dr))*

plantarlabb-vacío. *e.* abb-vacío) plantar!/;, *e. dr)*

plantar(insertar(e. *iz). f. dr) <= e < f*

plantar!/;, *f.* insertarte, *dr)) s= e > f*

La especificación de la operación está? sigue la misma estructura: se distingue si el árbol es vacío o no, y en el segundo caso se distingue si el elemento buscado está en la raíz, o es menor o mayor que la raíz, para buscar de forma recursiva en el hijo izquierdo o derecho, respectivamente.

está?(e, abb-vacío)	falso
está?(e. plantar!/;. *e. dr))*	cierto
está?(e. plantar!/;, *f. dr)}*	está?(e, *iz) <= e < f*
está?(e. plantar!/;, *f. dr))*	está?!/'. *dr) <= e > f*

Las operaciones mínimo y máximo son parciales, no estando definidas para el árbol vacío. La definición es sencilla: en un árbol binario de búsqueda el mínimo es el elemento *más a la izquierda* en la representación gráfica usual de un árbol, y el máximo es el elemento *más a la derecha.*

mínimo(abb-vacío) = error
mínimo(plantar(abb-vacío. *e. dr))* = *e*
mínimo(plantar(/;, *e. dr))* —
mínimo!/;) <=¬es-abb-vacio?(/;)
máximo(abb-vacio) = error
máximo(plantar(rz, *e.* abb-vacio)) = *e*
máximo(plantar(íp, *e. dr))* =
máximo(rfr) <= ^es-abb-vacio?(<7r)

La operación eliminar tiene que eliminar del árbol el nodo con el elemento a eliminar, si este está, y dejar el árbol sin modificar si no está. Además, el árbol resultante tiene que seguir siendo de búsqueda. Hay diferentes formas de especificar esta operación, y el árbol resultante depende de la elección tomada. Nosotros seguimos la siguiente alternativa. Borraren un árbol vacío no modifica el árbol. Si el elemento a eliminar está en la raíz, se distinguen casos según los hijos sean vacíos o no. Si el hijo izquierdo es vacío nos quedamos con el derecho, y viceversa. Cuando ninguno de los hijos es vacío, para mantener la propiedad de que el árbol sea de búsqueda se coloca el elemento mínimo del hijo derecho en la posición que ocupaba la raíz (elemento

eliminado), y se elimina recursivamente dicho elemento mínimo del hijo derecho. Si el elemento a borrar no está en la raíz, se pasa a borrar en el hijo izquierdo o en el hijo derecho, dependiendo de la relación de orden entre el elemento a borrar y la raíz.

eliminarle. abb-vacio) = abb-vacio
eliminarle, plantarfíz, *e.* abb-vacio)) =
íz
eliminarle, plantarfabb-vacío. e, *dr)) =*
dr
eliminarle. plantar(zc. e. *dr))* =
plantarfíz, mínimo(r/r).
eliminar(minimo(í/r), *dr))*
<= -'es-abb-vacío?(íp) A -
■es-abb-vacio'^{:>}(í/r) eliminarle,
plantario, *f. dr))* = plantar(eliminar(e,
íz). *f. dr)) <=* e < *f*
eliminarle, plantar!/-, *f. dr))* =
plantarfíz, *f,* eliminarle, *dr)) <=■ e> J*

Una posible alternativa para especificar esta operación sería, en el caso en que los dos hijos fueran no vacíos, poner en la raíz el máximo del hijo izquierdo, borrándolo de dicho hijo Esto no significa que las dos especificaciones sean equivalentes, como ha ocurrido otras veces cuando veíamos diferentes posibilidades para especificar una misma operación. En este caso, se especificarían operaciones *diferentes* ya que el resultado puede no ser el mismo utilizando una u otra alternativa.

Por último, la operación es-abb-vacío? distingue casos según las constructoras.

es-abb-vacio?(abb-vacío) = cierto
es-abb-vacio? (plantarfíz, *e, dr))* = falso
fespecificación
Desarrollar una implementación dinámica de los árboles binarios de búsqueda con las operaciones especificadas en el Ejercicio 7.1.
------------------Solución----------------
La estructura enlazada es análoga a la utilizada en el Ejercicio 6.3 para los árboles binarios (véase la Figura 6.2).
tipos
enlace-abb = **puntero a** *nodo-abb*
nodo-abb — reg
valor : elemento
iz, dr : enlace-abb
freg
árbol-bb = enlace-abb
ftipos
El árbol vacío está representado por el enlace nulo.
fun abb-vaoíoO **dev** *a : árbol-bb* { 0(l) }
a := nil
ffun
Al insertar un elemento se distinguen casos según el árbol sea vacío o no (igual que hacen las ecuaciones dadas en la especificación del Ejercicio 7.1). Si es vacío, se reserva espacio de memoria para un nuevo nodo, donde se almacena el elemento insertado, y se hacen vacíos el hijo izquierdo y el derecho (inicia-lizándolos a enlaces nulos). Si el árbol no es vacío, se compara el elemento a insertar con el elemento en la raíz. Si son iguales, no se hace nada. En caso

contrario, se hace una llamada recursiva para insertar en el hijo izquierdo o en el derecho. Obsérvese cómo, en estos dos últimos casos, el nuevo nodo quedará enlazado como hijo izquierdo o derecho de algún nodo ya presente en el árbol, al ser *a* un parámetro de entrada/salida.

proc insertarle *e : elemento, a : árbol-bb)* { O(altura(o)) }
  si *a* = nil **entonces**
    reservaría)
    a f *.valor := e*
    a f *iz* := nil ; *a* f *.dr* := nil
  si no
    casos
       e = *a* j *.valor* —* **nada**
      0 *e* < *a* f *.valor* —> *insertarle, a* f .íz)
      0 *e* > *a* "T *valor* —> insertarle, *a* f *.dr)*
    fcasos
  fsi
fproc

Aunque su uso no es habitual, también implementamos la operación plantar, ya que resultará útil en ciertos casos (véanse, por ejemplo, los Ejercicios 7.4 y 13.7). La operación tiene sentido solamente cuando el elemento a colocar en la raíz es mayor que todos los elementos en el hijo izquierdo y menor que los del hijo derecho, como requiere la precondición del algoritmo.

((es-abb-vacio?(zz) v máximo(íz) < *e)* A (es-abb-vacío?(Jr) v e < mínimo(Jr)) |
fun plantarl/.- : *árbol-bb. e : elemento, dr : árbol-bb)* **dev** *a : árbol-bb* **(0(l))**
  reservaría)

a'[.valor := e
a f *.iz := iz : W[.dr := dr* **ffun**

La implementación de la operación está? sigue la misma idea recursiva de las ecuaciones presentadas en la especificación del Ejercicio 7.1, al igual que la implementación de las operaciones mínimo y máximo.

fun está[9](í' : *elemento, a : árbol-bb)* **dev** *b : bool* (0(altura(<z)) }
si *a* = nil **entonces** *b* := falso
si no
casos
e— *a*f *.valor* —> *b* := cierto
0 *e* < *a[.valor* —> *b* -.= está?(e. *a* f ./;)
□ *e* > *a*f *.valor* —> *b* *:*= está?(e. *a* f *.dr)*
fcasos
fsi
ffun

fun mínimo(<7 : *árbol-bb)* **dev** *e : elemento* (©(altura(fl)) }
  si *a* = nil **entonces** error(Árbol vacío)
  si no
    si *a* f *.iz* = nil **entonces** *e := a* f *.valor*
    si no *e* := mínimo(n f *.iz)*
    fsi
  fsi
ffun
fun máximofo : *árbol-bb)* **dev** *e : elemento* { ©(altura(n)))
  si a = nil **entonces** errorfÁrbol vacío)
  si no
    si *a* f *.dr* = nil **entonces** *e := a [.valor*
    si no *e* := máximo^ f *.dr)*
    fsi
  fsi ffun

Al eliminar un elemento, solo se hace algo si el árbol no es vacío. En ese caso, si el elemento a eliminar es menor o mayor que la raíz, se sigue de forma recursiva con el correspondiente lujo Cuando el elemento a eliminar está en la raíz, se distinguen casos según los hijos sean vacíos o no. Si alguno es vacío, nos quedamos con el otro hijo. El caso en el que ninguno es vacío es algo más complejo. Las ecuaciones dadas en la solución del Ejercicio 7.1 especifican que hay que colocaren la posición de la raíz el elemento menor del hijo derecho, borrándolo a su vez. En vez de llamar a la función que consulta el mínimo y después hacer una llamada recursiva para eliminarlo (lo que supondría dos búsquedas del mínimo), el procedimiento

auxiliar eliminar-aux realiza todo el trabajo con un solo recorrido: busca el mínimo del hijo derecho, lo coloca en la raíz, y lo borra del hijo derecho. Obsérvese que este procedimiento es más eficiente, al buscar el mínimo una sola vez.

proc eliminarte *e : elemento, a : árbol-bb)* { 0(altura(«))) **var** *b : enlace-abb*
  si a 0 nil **entonces**
    casos
      e = a t *.valor* —>■
      casos
a f *.dr* = nil -> *b ;= a ; a := a* f *.iz ;* liberar(¿>)
        0 *a* T *-iz* — nil -> *b := a ; a := a* "j *.dr ;* liberar(b)
        0 *a* f *.dr*   nil A *a* f *.iz* nil —>■
        eliminar-aux(«, *a* f *.dr)*
      fcasos
     0 *e* < *a* t *.valor* —* eliminarte, *a* f *.iz)*
     0 *e* > *a* f *.valor* —> eliminarte, *a* "f *.dr)*
   fcasos
  fsi
fproc
{ *a* nil A *b* nil }
proc eliminar-aux(a, *b : árbol-bb)* { 0(altura(¿)) } **var** c : *enlace-abb*
  si *b* f *.iz* nil **entonces** eliminar-aux(a. *b* f *.iz)*
  si no
    a-[.valor := b[.valor
    c := b ; b := b[.dr
    liberar(c)
  fsi

fproc

El árbol de búsqueda es vacío cuando está representado por el enlace nulo.

fun es-abb-vacío?(« : ***árbol-bb)*** dev ***b : bool*** ((-9(1))

b := (a - nil)

ffun

En cuanto al coste de las operaciones, evidentemente abb-vacío y es-abb-vacío? tienen coste constante, al igual que plantar ya que esta operación enlaza árboles construidos como hijos del nuevo árbol, compartiendo la estructura enlazada, es decir, sin copias. El coste del resto de las operaciones depende de cómo estén repartidos los nodos del árbol entre los hijos. Todas las operaciones, en sus casos recursivos, hacen una llamada con uno de los hijos, por lo que el coste de las operaciones es proporcional a la altura del árbol. En el caso peor, cuando todos los nodos tienen solo un hijo no vacío y las llamadas recursivas recorren todos los nodos, el coste es lineal con respecto al número de elementos en el árbol.

Para evitar este caso degenerado existen implementaciones de los árboles de búsqueda que mantienen el árbol equilibrado. Como vimos en el Ejercicio 6.6. la altura de un árbol equilibrado de *n* nodos está en ®(log*n).* por lo que esta es la complejidad que se obtiene para las operaciones anteriores. Entre estas implementaciones podemos citar los *árboles AVL.* los *rojinegros* y los *2-3.*

sobre los cuales se puede obtener información en la mayoría de los libros de estructuras de datos, incluyendo (CLRS01. Fra94. HS94. WeiOO].

(a) Especificar e implementar de forma abstracta los tres *recorridos en profundidad* **de árboles binarios** de búsqueda.

(b) Escribir un algoritmo que ordene una lista (de elementos todos distintos) utilizando como estructura auxiliar un árbol binario de búsqueda.

-----------------Solución-------------------

Apartado (a)

Como un árbol binario de búsqueda es un tipo particular de árbol binario. los recorridos en profundidad se definen e implementan de la misma manera. Para más detalles, véase el Ejercicio 6.9.

Apartado (b)-------------

El algoritmo consiste en recorrer la lista, insertando todos sus elementos (distintos) en un árbol de búsqueda. Una posibilidad para continuar consiste en ir sacando del árbol sucesivamente el elemento mínimo y añadiéndolo a la lista por el final.

proc ordenación-arbóreal (/ : *lista)* **var** *a : árbol-bb*
a := abb-vacío()
mientras -’es-lista-vacía?(/) **hacer**
.V := izquierdo(Z) : elim-izq(/)
insertaría-, *a)*
(mientras
I := lista-vacíaO

```
mientras -'es-abb-vacío?(n) hacer
  ,v := mínimo(u) ; eliminar(.v, a)
  añadir-der(/, .v)
(mientras
fproc
```

El coste del algoritmo depende de la implementación del árbol de búsqueda utilizada: más concretamente. depende de la altura que tiene el árbol en cada momento. Con la implementación vista en el Ejercicio 7.2. en el caso peor la altura puede ser lineal con respecto al número de nodos en el árbol, que llegará a ser igual al número de elementos *n* — longitud(í) en la lista a ordenar. Por tanto, con dicha implementación el coste de ordenación-arbóreal es 0(ir).

Supongamos que la operación insertar sobre el árbol de búsqueda se ha implementado de forma que se mantenga equilibrado, lo cual garantiza un coste para ella en 0(log/n), donde *ni* es el número de nodos del árbol en el que se inserta. En el primer bucle el número de nodos del árbol varía entre 0 y n = longitud(/). y el bucle se repite exactamente *n* veces. Sabiendo que el coste de las operaciones sobre la lista es constante, el coste de este bucle viene expresado por la siguiente suma:

N

E^lo=>-

Veamos cuál es el orden de complejidad de esta expresión. Por un lado tenemos que

n *n*

^2iogj < i°g" = "log't.

J=i ;=1

por lo que la suma está en *O(n* loga). Por otro lado se cumple que

n *n* *n*

, logj > > . log- > -log-,
» >r?i >=r?i " " "
por lo que el sumatorio está en Q (log ") = Q (n log *n).*

Suponiendo que eliminar también mantiene el árbol equilibrado, el análisis del segundo bucle es análogo, aunque ahora el tamaño del árbol varía de *n* a 0. Por tanto, el coste en tiempo de este algoritmo de ordenación está en 0(n log*n).* El coste en espacio adicional está en 0(/i), que es el espacio ocupado por el árbol.

Otra posibilidad consiste en recorrer el árbol resultante en *inorden,* utilizando el hecho de que el recorrido en inorden de un árbol binario de búsqueda es una lista ordenada.

proc ordenación-arbórea2(/ : *lista)* **var** *a : árbol-bb*
  a := abb-vacío()
  mientras -■es-lista-vacía?(/) hacer
    x := izquierdo(í) : elim-izqf/)
    insertaría-, *a)*
  fmientras
  / := inorden(a)
fproc

En cuanto al coste, el primer bucle coincide con el del algoritmo anterior, por lo que su coste está en 0(/i logn) si el árbol se mantiene equilibrado, mientras que el coste del recorrido es lineal respecto al número de nodos; así pues, el coste de esta segunda versión está también en &(*n* log *n).* Sin embargo, la constante multiplicativa será menor, por

ser la segunda fase más eficiente. El coste en espacio adicional está en 0(n). como en la primera versión.

Diseñar un algoritmo que a partir del recorrido en preorden de un árbol binario de búsqueda construya el correspondiente árbol.
¿Qué ocurre si en vez del recorrido en preorden se parle del recorrido en inorden?

------------------Solución----------------

Sea *Ipre* la lisia que contiene el recorrido en preorden de un árbol binario de búsqueda. Dicho árbol es vacío si y solo si la lista *Ipre* es vacía. En otro caso, el elemento más a la izquierda de la lista será la raíz del árbol; sea x = izquierdo(ípre). Como el árbol es de búsqueda, los elementos en el hijo izquierdo serán menores que x. En el recorrido en preorden, el hijo izquierdo se recorre de forma recursiva después de la raíz, por lo que en *Ipre.* después de x, todos los elementos menores que x formarán el recorrido en preorden del hijo izquierdo. Por último, en el recorrido en preorden se recorre el hijo derecho de forma recursiva, por lo que el resto de elementos en *Ipre* (mayores que x) formarán el recorrido en preorden del hijo derecho. Conociendo los recorridos de ambos hijos, podemos construirlos con sendas llamadas recursivas.

El siguiente algoritmo implemento estas ideas:

fun reconstruir-abb(//>re : *lista)* **dev «** : *árbol-bb* **var** íz. *dr : árbol-bb. Ipre-iz. Ipre-dr : lista*

si es-lista-vacía?*(Ipre)* **entonces** *a* :=

abb-vacío()
si no
Ipre-dr := copiar-lista(/pr<?)
x := izquierdo(/pr<?-</r) ; elim-izq(/pre-Jr)
Ipre-iz := lista-vacía()
(separar los menores de los mayores)
mientras -■es-lista-vacía?*(Ipre-dr)* Ac izquierdo(/pze-Jr) < x **hacer**
y := izquierdof/pre-r/r) ; elim-izq(/pre-Jr)
añadir-der(*Ipre-iz,* y)
fmientras
iz := reconstruir-abb(/pre-/'z)
dr := reconstruir-abb(//w-dr)
a plantad *iz.* x. *dr)*
anular-lista(//>/<'-/;) ; anular-lista(/pr<’-r/r)
fsi ffun

En el caso recursivo, el bucle que separa los menores de los mayores tiene un coste lineal respecto al número de elementos en la lista. La suma de las longitudes de las dos listas construidas es uno menos que la longitud de la lista original, porque no contienen el elemento a colocar en la raíz. Estas listas son las utilizadas en las llamadas recursivas, por lo que la recurrencia que expresa el coste en tiempo del algoritmo es

Ti |-| co " = 0
| *T(p)* + *T(q)* + *c/n n > 0*

donde *n* es el número de elementos en la lista argumento, y *p* + *q* = n — 1. Por tanto, como se vio en el Ejercicio 6.6. *T(n)* e ©(n^2). El coste en espacio

adicional es cuadrático debido a que se hacen copias en variables auxiliares de tipo *lista* en todas las llamadas recursivas.

Si en vez del recorrido en preorden partimos de una lista con el recorrido en inorden, el árbol de búsqueda no se puede reconstruir de forma unívoca, ya que cualquier árbol de búsqueda que contuviera exactamente los elementos de la lista daría lugar al mismo recorrido en inorden al consistir este en una lista ordenada. Por ejemplo, los dos árboles de búsqueda de la Figura 7.1 tienen como recorrido en inorden la lista [3, 5. 7], De hecho, dado un conjunto de elementos, existen muchos árboles de búsqueda diferentes con exactamente esos elementos, por lo que podría intentarse construir un árbol de búsqueda *óptimo* en cieno sentido. El Ejercicio 13.7 investiga una de estas posibilidades.

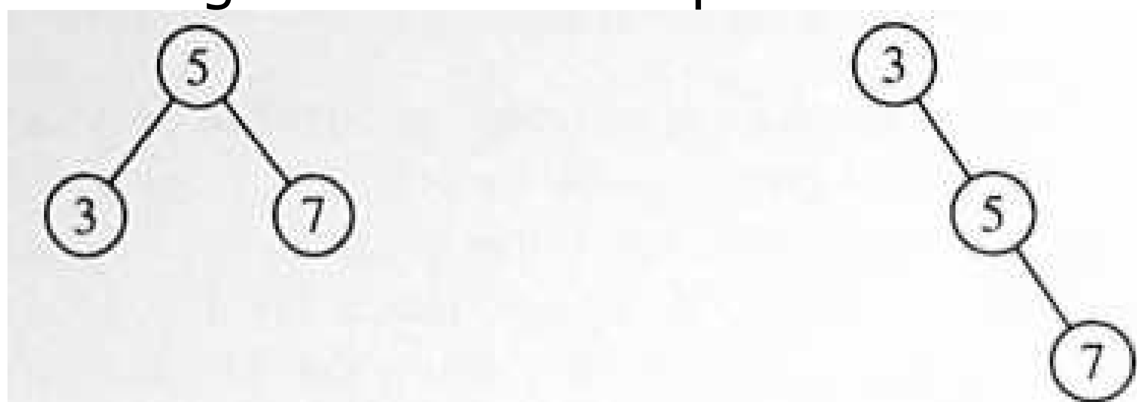


Figura 7.1: Dos árboles de búsqueda con el mismo recorrido en inorden.

(b) Especificar un TAD para describir árboles binarios de búsqueda en cuyos nodos se guardan dos elementos, una clave y un valor asociado a dicha clave, con las siguientes operaciones:

- crear un árbol binario de búsqueda vacío,
- insertar un valor asociado a una clave (en cada momento, el valor asociado a una clave es único),
- consultare! valor asociado a una clave.
- determinar si existe un nodo con una clave dada,
- consultar la clave mínima,
- consultar la clave máxima,
- eliminar el nodo que contenga una clave dada,
- determinar si el árbol es vacío.

(c) Modificar la implementación de árboles binarios de búsqueda del Ejercicio 7.2 para implementar estos nuevos árboles.

-----------------Solución------------------

Apartado (a)

Ya que las búsquedas en el árbol (para consultar o eliminar) se realizan utilizando una clave, el orden entre los nodos del árbol vendrá impuesto por el orden entre las claves. Por ello, definimos un parámetro que declara un tipo para las claves *(clave)* con operaciones de igualdad y orden total, junto con las propiedades que estas deben cumplir. Vamos a definir el parámetro en dos pasos, definiendo primero un parámetro de claves con igualdad, *CLAVES,* que se extiende a continuación con una

operación de orden en el parámetro *CLAVES<.* Lo hacemos así porque las claves con igualdad son útiles por sí mismas, como veremos en el Ejercicio 7.6. Nótese que el parámetro *CLAVES* es esencialmente el mismo que *ELEM=* definido en la Sección 1.1.5, solo que ahora el tipo definido se llama *clave* en vez de *elemento.*

parámetro *CLAVES*
usa *BOOLEANOS*
tipos *clave*
operaciones
_ == _ : clave clave —> bool
_ : clave clave —> bool
variables
c.d.e : clave
ecuaciones
(x == *y)* = cierto <= x = y
x = *y* <= *(x* == y) = cierto
x / y = -"(x == y)
fparámetro

parámetro *CLAVES*< **usa** *CLAVES*
operaciones
_ < _ : clave clave —» bool
_ > _ clave clave —>■ bool **variables**
c. d. e : clave **ecuaciones**
c < *c* =falso (antirreflexividad)
c < *e* =cierto <=*c* < *d* A *d* < *e* (transitividad |
c < *d v d* < *c* — cierto <= *c d* { totalidad j
c > *d* = *d* < *c*
fparámetro

En los árboles de búsqueda especificados en el Ejercicio 7.1. cuando se inserta un elemento que ya está en el árbol, este no se modifica. En esta nueva versión, podemos insertar un valor asociado a una clave que ya está en el árbol, posiblemente asociada a otro valor. Para no decidir ahora si es el primer valor el que permanece, o el segundo, o una combinación de ambos, vamos a exigir al parámetro de los valores que tenga una operación combinar que defina, a partir del valor actual y del nuevo, cuál es el valor que permanece en el árbol asociado a la clave correspondiente.

parámetro *VALOfíES-MODIFICABLES*
tipos *valor*
operaciones
combinar *valor valor —> valor*
fparámetro

Para especificar este nuevo tipo de

árboles de búsqueda seguimos las ideas utilizadas para especificar árboles binarios de búsqueda en el Ejercicio 7.1. aunque ahora plantar e insertar tienen un argumento más y las búsquedas se harán por claves La operación insertar ahora utiliza combinar cuando en el árbol ya hay un valor asociado a la clave que se está insertando; y la definición de la nueva operación consultar sigue la estructura de está[9], solo que en vez de un booleano se devuelve un valor. Para más explicaciones consúltese el Ejercicio 7.1. Obsérvese que los nombres de las operaciones son los mismos que los utilizados en dicho ejercicio.

especificación *ARBOLES-DE-BÚSOUEDA-CON-PARES[CLAVES<, VALORES-MODIFICABLES]*
usa *BOOLEANOS*
tipos *árbol-bb*
operaciones
abb-vacío : —>*árbol-bb* (constructora)
plantar : *árbol-bb clave valor árbol-bb* —*árbol-bb* { constructora j
insertar *clave valor árbol-bb* —> *árbol-bb*
consultar *clave árbol-bb* —*valor*
está? *clave árbol-bb* —► *bool*
mínimo *árbol-bb* —>•,> *clave*
máximo *árbol-bb* —► *p clave*
eliminar *clave árbol-bb* —> *árbol-bb*
es-abb-vacío? *árbol-bb* —> *bool*

variables
c.d : clave
v, w : valor
iZ, dr . árbol-bb

ecuaciones plantarle, *c.v.dr)* = error
<= ->(es-abb-vacío?(íz) v *c* > máximo(íz)) v
-'(es-abb-vacío?(dr) v *c* < mínimo(r/r))
insertarte, ti, abb-vacío) =
plantar(abb-vacío, c, v. abb-vacío)
insertarte, u, plantarte, c. *w. dr))*
= plantarfíj, *c.* combinar(w, *v). dr)*
insertarte, *v.* plantarte,*d.w, dr))* =
plantar(insertar(c, *v. iz), d, w. dr)* <= *c* < *d*
insertarte, u. plantarte:,*d,* u>.
dr)) —plantarfíz. *d, w.*
insertarte, *v, dr))* <= c> *d*
consultarte, abb-vacío) = error
consultarte, plantarte:, c. i>. *dr))* =
v
consultarte, plantar (;;, *d, w, dr))* =
consultarte,*iz)* <= *c* < *d*
consultarte, plantarte, *d, w, dr))* =
consultarte,*dr)* <= c >
d
está?(c. abb-vacío) = falso
está?(e, plantarte, e, v. *dr))* =
cierto
está?(c, plantarte, *d, w. dr))* =
está?(c,é)<= c < *d*
está?(e. plantarle, *d. w. dr))* =

está?(c,*dr)* <= *c* > *d*
mínimo(abb-vacío) = error
mínimo(plantar(abb-vacío, e. *v. dr))* =
c mínimo(plantar(é, c. *v. dr))* =
mínimo(íz) <= --es-abb-vacíoVté)
máximo(abb-vacío) = error
máxlmo(plantar(é. c, v. abb-vacío)) =
c máximo(plantar(é, c, *v, dr))* =
máximo (dr) <= -■es-abb-vacío? *(dr)*
eliminarte, abb-vacío) = abb-vacío
eliminarte, plantarte, c, v. abb-vacío))
= é eliminarte, plantar(abb-vacío. e,
v. *dr)) = dr* eliminarte, plantarte, c, *v. dr))* = plantarte, mínimo(rfr),
consultar(mínimo(r//')).
eliminar(mínimo(r/?'),
dr))
4= -■es-abb-vacío?(iz) A -
'es-abb-vacío?(</r) eliminarte,
plantarte, *d. w. dr))* =
plantar(eliminar(c, *iz). d. w. dr)* <= *c*
< *d*
eliminarte. plantar(r'z, *d. w, dr))* =
plantarte, *d, w.* eliminarte, *dr))* <= **e** > *d*
es-abb-vacío? (abb-vacío) = cierto
es-abb-vacío?(plantar(iz, c, *v. dr))* =
falso

fespecificación

Apartado (b)---------------------

Para implementar los nuevos árboles utilizamos una representación muy similar a la utilizada en el Ejercicio 7.2, aunque ahora los nodos en vez de un elemento tienen una clave y un valor.

tipos

enlace-abb = **puntero a** *nodo-abb*
nodo-abb = **reg**
clave : clave
valor: valor iz. dr : enlace-abb
freg
árbol-bb — enlace-abb
ftipos

Las operaciones abb-vacío y es-abb-vacío? no se modifican. Las operaciones está?, mínimo, máximo

y eliminar son prácticamente iguales, solo que ahora reciben como argumento una clave en vez de un elemento, y las comparaciones se hacen por claves.

La operación insertar se diferencia de la implementada en el Ejercicio 7.2 en que cuando se inserta un valor asociado a una clave que ya existía se utiliza combinar. Suponemos que la operación combinar se ha implementado como un procedimiento que modifica su primer argumento (el valor que había en el árbol) combinándolo con su segundo argumento (el nuevo valor).

proc insertarle *c : clave,* **e** *v : valor, a : árbol-bb)* (C-)(altura(<;)) }
  si *a* = nil **entonces**
    reservarte;)
    a í *.clave :=* c : *a* j *.valor :=* u ; *a* j *.iz* '■= nil ; *a* f *.dr* := nil
  si no
    casos
      c= er f *.clave* —> combinarte; f *.valor,* v)
      ▫ *c* < *a* f *.clave* —> insertarte, v. *a* f *.iz)*
      ▫ *c* > *a* T *.clave* —> insertarte, v. *a* j *.dr)*
    fcasos
  fsi
fproc

Al igual que en las ecuaciones, la implementación de la operación consultar es muy similar a la de está?, aunque en este caso se devuelve un valor.

fun consultarte : *clave, a : árbol-bb)* **dev**

v : *valor* (0(altura(e;)) |
si *a* = nil **entonces** error(Clave no definida)
si no
casos
c= *a*j *.clave* —> v := *a* f *.valor*
▫ < < e;t .c/e;ve —> *v* :=
consultarte, *a* f *.iz)*
D < > *a*j *.clave* —> *v* :=
consultarte, *a* f *.dr)*
fcasos
fsi
ffun

Especificar un TAD para describir las tablas con claves y valores pertenecientes a tipos dados como parámetro, y las siguientes operaciones:

- crear una tabla vacía.
- insertar un valor asociado a una clave, o actualizarlo si la clave ya tiene asignado un valor,
- determinar si una clave tiene asociado algún valor,
- consultar el valor asociado a una clave.
- eliminar el valor asociado a una clave, y
- determinar si la tabla es vacía, es decir, no hay ninguna asociación entre claves y valores.

-------------------Solución--------------------

Utilizamos el parámetro *CLAVES* dado en la solución del Ejercicio 7.5 porque las claves tienen que tener definida una operación de igualdad (pero no es necesaria una operación de orden) y el parámetro *VALORES-MODIFICABLES* dado en la misma solución, porque aquí también los valores asociados a una clave son *inodificables.*

Como constructoras elegimos la operación que crea la tabla vacía (tabla-vacía) y la operación para insertar asociaciones entre claves y valores (insertar-tabla). La operación para consultar el valor asociado

a una clave, consultar-tabla, es parcial, estando definida solo en el caso de que la clave tenga ya asociado algún valor en la tabla.

especificación *TABLAS[CLAVES, VALORES-MODIFICABLES]*
usa *BOOLEANOS*
tipos *tabla*
operaciones
tabla-vacía: —> *tabla* (constructora)
insertar-tabla *:* *clave valor tabla* —► *tabla* (constructora)
está-tabla? *: clave tabla* ▪—> *bool*
consultar-tabla : *clave tabla* -—► *p* *valor*
eliminar-tabla *:* *clave tabla* —> *tabla*
es-tabla-vacía? : *tabla* —> *bool*
variables
c, d : clave
v. u> : valor t : tabla

Las constructoras *no son libres* ya que insertar dos valores asociados a la misma clave es equivalente a insertar la combinación de ambos en el orden apropiado: el valor que estaba en la tabla se combina con el nuevo. Además, el orden en el que se insertan valores asociados a claves diferentes no es importante. Las siguientes ecuaciones de equivalencia entre términos construidos reflejan ambos casos.

ecuaciones
insertar-tablaíc. *w.* insertar-tablaíc, v, r)) = insertar-tablaíc, combinarfu. w), Z)

insertar-tablaíc/. ic, insertar-tablaíc. v, í)) = insertar-tablaíc, *v.* msertar-tablafr/. u-, *I)) <= c^d*

El resto de las operaciones se definen distinguiendo casos según las constructoras y utilizando la operación de igualdad entre claves.

está-tabla?(c. tabla-vacía) = falso

está-tabla?(c. insertar-tablaíc/, *v,/)) = c* == *d* V está-tabla?(c, r)

Para consultar el valor asociado a una clave hay que tener en cuenta que se han podido insertar varios valores para una misma clave, por lo que en ese caso habrá que devolver la combinación de todos ellos. Esto es debido a que las ecuaciones tienen que ser independientes del representante elegido.

consultar-tablafc,/) = error <= --está-tabla?(c, *t)*

consultar-tabla(c. insertar-tablaíc, v./)) = v <= --está-tabla? (c./)

consultar-tablafc, insertar-tablaíc, v. r)) = combinar(consultar-tabla(c. r). *v) <=* está-tabla? (c, Z) consultar-tablaíc. insertar-tablaíc/. u. Z)) = consultar-tablafc, z) <= *c d*

Obsérvese cómo, al eliminar una clave, hay que eliminar *todas* las apariciones de dicha clave en la tabla, y de ahí la recursión en la parte derecha de la segunda ecuación para eliminar-tabla.

eliminar-tablafc, tabla-vacía) = tabla-vacía

eliminar-tabla(c, insertar-tablaíc. v, z)) = eliminar-tablaíc, r)

eliminar-tablafc. insertar-tablaíc/, v, z))
= insertar-tablaíc/, v. eliminar-tablaíc, r))
<= c # *d*
es-tabla-vacía?(tabla-vacía) = cierto
es-tabla-vacía?(insertar-tabla(c. u./)) =
falso
fespecificación

En el caso de que las claves de una tabla tengan definida una relación de orden total, especificar una operación que devuelva la lista de pares *{clave, valor}* que se encuentran insertados en la tabla, ordenada por claves.

-----------------Solución---------------

Según la especificación de las tablas del Ejercicio 7.6. el orden en el que se insertan valores asociados a claves diferentes no es relevante, como muestra la segunda ecuación de equivalencia dada en la solución. Por tanto, no existe ningún criterio para enumerar *seciiencialmeiite* los elementos de una tabla en una lista. Al no existir una forma única de poner “en orden" en una lista los elementos de la tabla, la definición de una operación que devolviera dicha lista igualaría listas en principio diferentes. Por tanto, la operación no tiene sentido si los elementos no se pueden ordenar en algún sentido, por lo que, como pide el enunciado, especificaremos la operación cuando las claves tienen orden.

Entonces sí podemos enumerar los elementos en una lista de forma única, por ejemplo, de menor a mayor clave. Para definir la operación utilizamos las listas ordenadas especificadas en el Ejercicio 5.7. y la operación insertar-ord que inserta, manteniendo el orden, un elemento en una lista ya ordenada. Para instanciar las listas ordenadas especificamos un parámetro que define pares (*clave, valor)* y una operación de

orden entre pares.

parámetro *PARES-ORDENABLES*
usa *CLAVES<. VALORES-MODIFICABLES* **tipos** *par* **operaciones**
( ) clave valor —> par
_ < _ clave clave —► bool
variables
c. d : clave
v.w : valor
ecuaciones
(c. v) < (d. w) = c < d
fparámetro

especificación *TABLAS-ORDENADAS[CLAVES<, VALORES-MODIFICABLES}*
usa *TABLAS[CLAVES<, VALORES-MODIFICABLES}. LISTAS-ORDENADAS[PARES-ORDENABLES]*
operaciones
recorrer-ordenada : *tabla —> lista[par]*
variables
c : clave
v : *valor*
t : tabla

Ya que en la tabla se puede haber insertado varias veces información asociada a una misma clave, tendremos que utilizar combinar para obtener el valor final asociado a una clave que se añade a la lista. Y para que no haya pares en la lista con la misma clave, al hacer la llamada recursiva hay que borrar antes la clave insertada.

ecuaciones
recorrer-ordenada(tabla-vacía) = [1
recorrer-ordenada(insertar-tabla(c, u,

r)) = insertar-ord((c. v). recorrer-ordenada(r))

<= -■está-tabla? (c, r)

recorrer-ordenada(insertar-tabla(c. *v. t))* = insertar-ord((c, combinar(consultar-tabla(c. r). r)). recorrer-ordenadateliminar-tablatc, *t)))*

<= está-tabla?(c./)

(especificación

Cuando las claves de una tabla tienen orden, esta puede implementarse mediante un árbol de búsqueda **que almacena en** sus nodos parejas de tipo (*clave, valor). *Implementar las tablas con las operaciones **especificadas en los** Ejercicios 7.6 y 7.7 utilizando árboles de búsqueda.

-------------------Solución-----------------

Los árboles de búsqueda especificados e implementados en el Ejercicio 7.5 están especialmente diseñados para implementar con ellos tablas cuyas claves tengan orden, por lo que la implementación de cada operación es realmente sencilla, reduciéndose a una llamada a la correspondiente operación de los árboles. El coste de las operaciones depende de la implementación de los árboles de búsqueda utilizados En los costes siguientes, suponemos que se utilizan árboles *equilibrados* (AVL. por ejemplo). El tamaño n de los datos es el número de asociaciones entre claves distintas y valores en la tabla.

tipos
tabla = *árbol-bb*
ftipos
fun tabla-vacía() dev *t* : *tabla* (0(l))
t árbol-vacío()
ffun
proc insertar-tablaje *c* : *clave,* e v : *valor, t* : *tabla)* { Qjlog *n)*) insertarte, v. *t)*
fproc

fun está-tabla?(c : *clave, t : tabla)* **dev** *b : bool* { Ojlogn))
b := está?(c, *t)*
ffun
fun consultar-tablaje : *clave, t : tabla)* **dev** *v : valor* (©jlog/i))
v := consultarte, r)
ffun
proc eliminar-tablaje c : *clave, t : tabla)* { Qjlogn))
eliminarte, *t)*
fproc
fun es-tabla-vacía?*(i : tabla)* dev *b : bool* (0(l))
b es-abb-vacío?(r)
ffun

Para recorrer la tabla de forma ordenada podemos utilizar el recorrido en *inorden* del árbol de búsqueda (véase el Ejercicio 7.3), ya que este recorrido está ordenado por claves.

fun recorrer-ordenadajr : *tabla)* **dev /:** *!ista[par]* (0(/i))
/ := inordenjr)
ffun

Implementar la especificación de multiconjuntos del Ejercicio 1.8 en el caso en el que los elementos tengan una relación de orden, en términos de tablas donde los elementos del multiconjunto son las claves y las multiplicidades son los valores asociados.

-----------------Solución------------------

La operación de combinación sobre las multiplicidades (véase el Ejercicio 7.5) es la suma de números. Aunque las multiplicidades siempre van a ser positivas, para quitar una aparición de un elemento que está en el multiconjunto vamos a insertar de nuevo el elemento con valor asociado -1, de tal forma que se *reste* 1 a su multiplicidad. Por tanto, utilizaremos los elementos como claves y los números enteros como valores. Además, para realizar operaciones como la unión de multiconjuntos será necesario poder recorrer todos los elementos de un multiconjunto. por lo que utilizaremos tablas ordenadas (véase el Ejercicio 7.7). Al recorrer las tablas, las listas ordenadas devueltas serán de pares *(elemento, multiplicidad)* de tipo *par-mcjto.*

tipos

multiconjunto = *tablajelemento. ent]*

par-mcjto **= reg**

elem : elemento

niult : ent

freg
ftipos
proc combinarfm] : *ent.* **e** nu *: ent)*
ni | := ni , + 1112
fproc

No conviene que la multiplicidad de un elemento en la tabla sea 0. pues entonces el tamaño de la tabla no tendría relación con el tamaño del multiconjunto; por ejemplo, el multiconjunto vacío se podría representar con una tabla arbitrariamente grande cuyas multiplicidades fueran todas 0.

El coste de las operaciones será proporcional al coste de las operaciones sobre las tablas utilizadas, y estas dependerán, a su vez. de la implementación de las tablas. Supondremos que las tablas están implementadas con árboles de búsqueda (véase el Ejercicio 7.S) *equilibrados,* por lo que al hablar de costes haremos referencia a los costes en tal situación.

La operación constante mcjto-vacío corresponde directamente a la tabla vacía.

fun mcjto-vacío() **dev** ,v : *multiconjunto*
{ 0(1))
i := tabla-vacíaO
ffun

Para implementar añadirte, A). añadimos a la tabla el elemento *e* con multiplicidad 1. Si el elemento *e* ya está con multiplicidad asociada *ni.* como la operación combinar es la suma, la multiplicidad se incrementa a *ni* + I. Si

el elemento no está, quedará insertado con multiplicidad 1. El coste está en 0(log *n).* donde *n* es el número de nodos en el árbol, que coincide con el número de elementos distintos en el multiconjunto.

proc añadirle *e : elemento,* .v : *multiconjunto)* (0(logii))
  insertar-tabla(e. 1,A)
fproc

Para implementar múltiple, A). buscamos *e* en la tabla que representa el multiconjunto A. Si no está, el resultado es 0. Si ya está, consultamos su multiplicidad *ni.* que es el resultado.

fun múltiple : *elemento,* A *: multiconjunto)* **dev** *ni : ent* (0(logn))
  si está-tabla?(e. A) **entonces** *ni* := consultar-tabla(e. A)
  si no *ni* **:= 0**
  fsi
ffun

I "*i* > 0 }

El predicado es-mcjto-vacío? corresponde directamente al predicado semejante sobre tablas.

fun es-mcjto-vacío?(x : *multiconjunto)* dev *b : bool* { 0(l))

b := es-tabla-vacía?(A) **ffun**

Para implementar quitarje, x), buscamos *e* en la tabla que representa el multiconjunto x. Si no está, no hacemos nada más pues la tabla se deja igual. Si ya está, consultamos la multiplicidad *m.* Si *ni* > 1, la modificamos poniendo en su lugar *ni* — 1, al insertar en la tabla *e* con valor — 1. Si *ni* = 1, como no puede quedarse en 0, hay que borrar el elemento de la tabla.

proc quitar(e *e : elemento,* x : *multiconjunto)* (©jlogn))

si está-tabla?(c, x) entonces

m consultar-tablaje, x)

si *m* > 1 **entonces** insertar-tablaje, — l,x)

si no eliminar-tablaje, x)

fsi

fsi

fproc

La operación borrar corresponde directamente a la operación semejante sobre tablas. En particular, cuando el elemento no está, la tabla se deja igual.

proc borrarje e : *elemento,* x : *multiconjunto)* { ©jlogn))

eliminar-tablaje, x)

fproc

Para implementar la unión de multiconjuntos. se recorren los elementos de cada uno de los multiconjuntos dados como argumentos

(recorriendo la lista obtenida utilizando la operación recorrer-ordenada de tablas, véase el Ejercicio 7.7) para añadirlos en el multiconjunto resultado.

fun uniónjx, *y : multiconjunto)* dev z : *multiconjunto*
var / : *listajpar-mcjtoj*
Z := tabla-vacíaj)
/ := recorrer-ordenadajx)
mientras -■es-lista-vacía?(/) hacer
p izquierdoj/) ; elim-izq(Z)
insertar-tablajp.e/eni, *p.niult,* z)
fmientras
/ := recorrer-ordenadajy)
mientras -■es-lista-vacíaVj/) **hacer**
p := izquierdoj/) ; elim-izqj/)
insertar-tablajp.e/e/n, *p.niult,* z)
fmientras ffun

Si «i y *112* son los tamaños de los argumentos de la función unión, el coste total de los dos recorridos está en Ojni 4- *112).* El tamaño del resultado se va incrementando a medida que se van añadiendo elementos, llegando a ser, en el caso peor, igual a *ni +112-* Como cada inserción tiene coste logarítmico con respecto al tamaño, según el análisis detallado en el Apartado (b) de la solución del Ejercicio 7.3. el coste total de las inserciones está en Ojjn, 4-/12) logjn 1 4-*112)).* que coincide con el coste de unión al ser mayor que el coste de los recorridos.

Para implementar la intersección de multiconjuntos, seguimos la misma

idea que para la unión: recorremos el primer argumento e insertamos en el resultado aquellos elementos que también están en el segundo argumento, con la multiplicidad igual al mínimo de las multiplicidades en los argumentos. **fun** intersección(J:, y : *multiconjunto)* **dev** *z : multiconjunto*
var / : *lista[par-mcjto]*
Z := tabla-vacía!)
I := recorrer-ordenada(.v)
mientras ~-es-lista-vacía?(/) hacer
p := izquierdo!/) ; elim-izq(/) **si** está-tabla? (/xe/ent. y) **entonces** (pertenece **a** la intersección)
ni := consultar-tablai p.e/cw, *y)*
insertar-tabla(/;.<'/e»i. mín(/n. *p.mult),* z)
fsi
fmientras
ffun

Haciendo un análisis similar al de la unión, como se recorre el primer argumento de tamaño n,. se busca y consulta en el segundo de tamaño /n, y se inserta en el resultado cuyo tamaño es como mucho min(/í |. *ni),* el coste queda en 0(ni logas)-

La implementación de la diferencia de multiconjuntos es completamente análoga a la de la intersección pero restando multiplicidades en vez de calculando el mínimo, por lo que el coste es el mismo.

fun diferencíala. y : *multiconjunto)* **dev** z : *multiconjunto* { ©(«i logztj)) **var / :**

lista[par-mc/to]
z := tabla-vacial)
/ .= recorrer-ordenada(.v) **mientras** -'es-lista-vacía?(/) **hacer** *p* := izquierdol/) . elim-izq(Z) **si** está-tabla?(p.e/cvn, y) **entonces**
ni = consultar-tabla(p.e/e»i. y)
si *ni* < *p.mult* **entonces** insertar-tabla! p.e/e/n. *p.mult — m.* z) **fsi si no**
insertar-tabla! p.e/eza. *p.mult.* z) **fsi**
fmientras
ffun

Para calcular cardinal(.v) y cardinal-dist(.v), hay que recorrer la tabla que representa el multiconjunto A . En el primer caso se van acumulando las multiplicidades que se encuentran en los nodos, mientras que en el segundo basta con contar los nodos (equivalentemente, sumar uno por cada nodo, o calcular la longitud de la lista con el recorrido).

fun cardinal(.v : *multiconjunto)* **dev** *m : nat (©(«))* **var / :** *listajpar-mcjtoj*
I := recorrer-ordenada(.v) ; *m* := 0
mientras --es-lista-vacía?!/) **hacer** *p* := izquierdo!/) : elim-izq(Z) *m := m +* *p.mult* **fmientras**
ffun
fun cardinal-distLv : *multiconjunto)* **dev** *m : nat* (©(n) } **var** *l : lista[par-mcjto]*
I := recorrer-ordenada(.v) *m* := longitud!/)
ffun

El *problema de las* ***concordancias*** consiste en, dado un texto, contar el número de veces que aparece en él cada palabra, y producir una lista ordenada alfabéticamente por palabras, donde cada palabra aparece acompañada del número de veces que ha aparecido en el texto. Suponiendo que el texto a analizar viene dado como una *lista de palabras,* siendo *palabra* un tipo disponible con su orden, desarrollar un algoritmo que resuelva el problema.

-----------------Solución-----------------

La idea de la solución que se propone es recorrer la lista dada de palabras e ir insertándolas en una tabla ordenada donde se utilizan las palabras como claves y el número de veces que una palabra ha aparecido como su valor asociado.

Como en el caso de los multiconjuntos (Ejercicio 7.9), la operación de combinación de valores se corresponde con la suma de números, pero ahora son naturales en vez de enteros. La lista ordenada (cuyos elementos son valores del tipo *par)* se obtiene recorriendo la tabla (con la operación recorrer-ordenada, véase el Ejercicio 7.7) una vez se han procesado todas las palabras.

tipos

par-concor **= reg**

pal: palabra

nuin : naf^{+}

freg

ftipos
fun concordancias(/ : *lista[palabrai)*
dev *Ip : lista[par-concor]* **var** *t : tabla[palabra, nar]. k : lista[palabra]*
si es-lista-vacía?(/) **entonces** *Ip* := lista-vacía()
si no
k := copiar-lista(/)
t := tabla-vacía()
mientras -■es-lista-vacia?(A) **hacer**
p := izquierdo(A) ; ellm-izq(A-)
insertar-tabla(/?. 1. /)
fmientras
Ip recorrer-ordenada (/)
anular-tabla(r)
fsi ffun

Suponiendo que la tabla está implementada mediante un árbol de búsqueda equilibrado (un árbol AVL, por ejemplo), el coste del algoritmo se calcula como sigue. La copia y el recorrido de la lista son lineales con respecto a la longitud de la lista; la inserción en el árbol de búsqueda es de coste logarítmico con respecto al número de nodos del árbol; y el recorrido en inorden del árbol es lineal con respecto al mismo número. En el caso peor, todas las palabras son distintas, por lo que el número de nodos del árbol coincide con la longitud de la lista inicial. Si tal longitud es *n.* el coste en el caso peor queda en *&(n* logn).

Dado un texto organizado por líneas, el *problema de las* ***referencias cruzadas*** consiste en producir un listado de palabras ordenado alfabéticamente, donde cada palabra del texto va acompañada de una lista de referencias que contiene los números de todas las líneas del texto en las que aparece la palabra en cuestión, con posibles repeticiones si la palabra aparece varias veces en una misma línea.
Suponiendo que el texto que hay que analizar viene dado como una *lista de listas de palabras,* siendo *palabra* un tipo disponible con su orden, desarrollar un algoritmo que resuelva el problema.

--------------------Solución----------------

Utilizaremos una tabla donde las claves serán las palabras y el valor asociado a cada clave será la lista de números (lista de referencias) donde la palabra ha aparecido. Como la lista resultado se obtendrá a partir de la tabla y tiene que estar ordenada por palabras, utilizaremos tablas con una operación de recorrido en orden (véase el Ejercicio 7.7).

tipos
  par-ref **= reg**
      pal : palabra
      refer : lista[nat]*
    freg
ftipos

La operación adecuada de combinación de valores consiste en añadir por la derecha a la lista que ya se encuentra en

el árbol el único elemento en la nueva lista, es decir, el número de línea de la nueva aparición de la palabra.

proc combinar(/| *: lista[nati.* **e** *I? : hsta[nat])*
añadir-der(/]. izquierdo^))
fproc

La función referencias-cruzadas distingue casos según el texto recibido como argumento (una lista de listas de palabras) sea vacío o no. Si es vacío, se devuelve la lista vacía. En otro caso, se inicializa una tabla auxiliar a vacía y se recorre el texto por líneas. Para cada línea (una lista de palabras) se recorren todas sus palabras, y cada palabra se inserta asociada a la lista unitaria con el número de línea correspondiente. Si estamos en la primera ocurrencia de la palabra, esta lista unitaria será la que se inserte en la tabla. Si la palabra ya había aparecido, la concatenación utilizada por combinar añadirá el nuevo número de línea al resto. Al acabar con el texto, la lista resultado se obtiene recorriendo *en orden* la tabla.

fun referencias-cruzadas!/ : *lista[lista[palabra]]***)** **dev** *Ip : lista[par-ref]*
var *i : tabla[palabra, lista[nat]l. k : lista[lista[palabra]]. línea : lista[palabra]*
si es-lista-vacía?(/) **entonces** *Ip* := lista-vacía()
si no
k .= copiar-lista(/)
i := tabla-vacíaO

número-línea := 0 (número de línea)
mientras -’es-lista-vacía?(Zr) **hacer** { recorre texto por líneas |
línea = izquierdo(Z.) : elim-izq(A')
número-línea := número-línea + I
mientras -^es-lista-vacia?(/mra) hacer (recorre línea por palabras) *p :*= izquierdo(/óteíí) ; elim-izq(/úie«)
insertar-tabla(/>, *urMaúa(núinero-línea). r)*
fmientras
fmientras
//> := recorrer-ordenada(t)
fsi
ffun

Suponiendo que la tabla está representada mediante un árbol de búsqueda equilibrado (un árbol AVL. por ejemplo), el coste del algoritmo se calcula como sigue. La copia y el recorrido de la lista de listas (los dos bucles **mientras** anidados) son lineales con respecto al número de palabras en el texto. La inserción en el árbol de búsqueda es de coste logarítmico con respecto al número de nodos del árbol, que. en el caso peor, cuando todas las palabras son distintas, coincide con el número total de palabras. Si tal número es *n.* como el recorrido en inorden del árbol es lineal con respecto al mismo, el coste en el caso peor queda en *(~)(n* logn).

Una *tabla dispersa (hash table.* en inglés) es una tabla implementada utilizando una función de dis**persión** *h*

: clave **—» 1** *..N* y un vector T[I ../V] que almacena las asociaciones entre claves y valores, **donde** las claves no necesariamente tienen orden. Cuando para dos claves distintas ci. *C2* ocurre que **/>(ci) =** *h(c2)* se dice que se ha producido una *colisión* y que las claves son *sinónimas.* Los diferentes tipos de tablas dispersas se diferencian en cómo se resuelven las colisiones.

(a) En una *tabla dispersa abierta* cada posición del vector almacena una *lista* de parejas de tipo *(clave, valor).* Las colisiones se resuelven almacenando todas las claves sinónimas *c* tales que /i(c) = *i* en la lista T[í). Implementar las tablas como tablas dispersas abiertas con las operaciones especificadas en el Ejercicio 7.6.

(b) En una *tabla dispersa cerrada* cada posición del vector almacena una pareja de tipo (*clave, valor).*

Las colisiones se tratan del siguiente modo: si el índice tj = *h(c)* asociado a una clave c produce colisión, se prueban otros índices L = prueba(2, c) ....í„ = pruebafm, c). etc., hasta que no se produzca colisión. Esta técnica se denomina *redispersión.* Implementar las tablas como tablas dispersas cerradas con las operaciones especificadas en el Ejercicio 7.6, suponiendo disponible la función de redispersión prueba.

-------------------Solución-------------------

Apartado (a)

Una posibilidad sería utilizar las listas de parejas *de forma abstracta,* dada una implementación de las listas con las operaciones necesarias. Dependiendo de las operaciones disponibles, esto podría tener como consecuencia que una búsqueda (por clave) seguida de una actualización realizara dos recorridos completos de una lista. Para evitar esto, en vez de utilizar listas de forma abstracta trabajaremos directamente sobre la estructura de enlaces. De esta manera, el tipo representante se define como sigue

tipos
enlace-tabla = **puntero a** *nodo-tabla*
nodo-tabla **= reg**
clave : clave
valor: valor
sig : enlace-tabla
freg
tabla **= vector [1 ,.N] de** *enlace-tabla*
ftipos

Crear la tabla vacía consiste en inicializar todas las listas a vacías.

fun tabla-vacia() **dev** *t : tabla (Q(N))*
r[l..N] := [nil]
ffun

Las operaciones insertar-tabla, está-tabla?, consultar-tabla y eliminar-tabla tienen todas algo en común en su comportamiento. Todas realizan una búsqueda en la lista que se encuentra

en la posición del vector indexada por el valor devuelto por la función de dispersión, y después, en caso de éxito, realizan una operación sobre el nodo encontrado de la lista. Esta operación de búsqueda vamos a implementarla mediante la función auxiliar localizar, que devuelve si existe en la estructura enlazada que comienza en un enlace dado como parámetro un nodo con la clave dada también como parámetro. En caso de éxito, también se devuelve un puntero al nodo con la clave. Para la operación de eliminación también es necesario un puntero al nodo anterior al que se borra, al no estar la lista doblemente enlazada. La función localizar devuelve también este valor, que es nulo si no hay anterior. El coste de localizar es lineal respecto al número de nodos de la lista.

fun localizare : *clave, p : enlace-tabla)*
dev (*encontrado : bool. ant. q : en'ace-tabla) encontrado* := falso
q := p ; ant := nil
mientras *q* nil A —•*encontrado* **hacer**
si *q* f *.clave = c* **entonces** *encontrado* := cierto
si no *ant := q ; q := q* f *.sig*
fsi
fmientras
ffun

Utilizando localizar la implementación del resto de las operaciones es muy sencilla. En todas se utiliza la función de dispersión h. Para insertar primero se busca la clave. Si se encuentra, se modifica el nodo encontrado para que almacene la combinación con el nuevo valor. Si no, se crea un nuevo nodo, que se añade (por la izquierda) a la lista que contiene las claves sinónimas.

proc insertar-tablafe *c : clave,* **e** u : *valor, t : tabla)* **var** *p. ant* ▪ *enlace-tabla*
i := h(c)
(encontrado, ant, p) := localizarte, *t*[í])
si *encontrado* **entonces** combinar(p f *.valor, v)* **si no**
reservar(p)
p f *.clave* := c; *p],valor := v ; p^.sig :*= Z[í]
/[/] := p
fsi
fproc

Suponiendo que el cálculo del valor de la función de dispersión tiene coste

constante, el coste del algoritmo anterior es proporcional al de la operación auxiliar localizar, que es lineal con respecto a la longitud de la lista r[z | donde se inserta. En el caso peor, la lista tendrá todos los elementos de la tabla, por lo que el coste será lineal con respecto al número de asociaciones entre claves y valores en la tabla. En la práctica, el coste dependerá de la bondad de la función de dispersión y de la tasa de ocupación de la tabla. Una buena función de dispersión debe ser uniforme, independiente de la apariencia de la clave, exhaustiva, y rápida de calcular. Si la función de dispersión es buena y la tasa de ocupación no es elevada, el coste de las operaciones puede ser constante. Para más detalles sobre funciones de dispersión y sus propiedades, consúltese, por ejemplo, el Capítulo 4 de [Fra94],

La implementación de las operaciones está-tabla? y consultar-tabla prácticamente consiste en una llamada a localizar. El análisis del coste es análogo al de insertar-tabla.

fun está-tabla?(c : *clave, l : tabla)* **dev** *b : bool* **var** *p, ant : enlace-tabla*
  i := h(c)
  (b, ant, p) := localizarte, z[i])
ffun
fun consultar-tabla(c : *clave, t : tabla)*
dev *v : valor*
var *p, ant : enlace-tabla*

i := h(c)
(encontrado, ant, p) := localizarte, z[í])
si *encontrado* **entonces** *v := p* t *.valor*
si no error(La clave no está)
fsi
ffun

Para eliminar, primero hay que localizar la clave. En caso de éxito, hay que distinguir casos según el nodo a eliminar sea el primero de la lista *(ant* = nil) o no, para saber si hay que modificar el enlace almacenado en la tabla, o un enlace en el nodo anterior. El análisis del coste es análogo al de insertar-tabla.

proc eliminar-tabla(e *c : clave, t : tabla)*
var *p, ant: enlace-tabla*
i **:= h(c)**
{encontrado, ant, p) := localizarte. r[í])
si *encontrado* **entonces**
si *ant* = nil **entonces** { borrar el primero de una lista 1
r[z] := *p'f.sig*
si no { saltar el nodo apuntado por *p*
) ant f *.sig := p* f *sig*
fsi
liberar(p)
fsi
fproc

La tabla es vacía si en todas las posiciones del vector hay listas vacías (enlaces nulos).

fun es-tabla-vacia?(/ : *tabla)* **dev** *v : valor* (0(N) }
b := cierto : í := 1
mientras *i* < *N A b* **hacer**
b := (r[í) = nil)
i := Í + 1
fmientras
ffun

Apartado (b)--------------------

Como indica el enunciado, una *tabla dispersa cerrada* se implemento mediante un vector en cuyas posiciones se almacenan asociaciones entre claves y valores (y más información, como veremos). La idea general intuitiva de las tablas dispersas cerradas es la siguiente. La tabla comienza con todas sus posiciones *libres.* Cuando se inserta la primera clave, se coloca en la posición indicada por la función de dispersión para esa clave. Al seguir insertando claves, se pueden producir colisiones, es decir, una clave debería colocarse en una posición ya *ocupada.* En ese caso, se “salta" por diferentes posiciones de la tabla, según indica la *función de redispersión* prueba, hasta encontrar una posición libre.

Cuando se busca una clave, también hay que seguir esta sucesión de saltos, hasta encontrar una posición con la clave buscada, o encontrar una posición libre, lo que indicaría que la clave buscada no está. Al borrar una clave, también se sigue la sucesión de saltos, y al encontrar la clave, se marca la posición como disponible, para que otra clave pueda ocupar esa posición. Sin embargo, la posición no puede marcarse directamente como libre, porque para encontrar claves que colisionaron con la ahora borrada, y por tanto se colocaron en posiciones siguientes (según los saltos), hay que seguir pasando por la posición que ahora queda “libre”. Para distinguir este hecho,

estas posiciones se marcarán como *borradas.* Estas posiciones se podrán utilizar para insertar nuevas claves y para “adelantar" (acercar a la posición devuelta por la función de dispersión) si es posible claves ya insertadas cuando se modifica su valor, pero tendrán que ser saltadas al buscar una clave.

El tipo representante para las tablas dispersas cerradas es el siguiente:

tipos
estado = (libre, ocupada, borrada)
triple = reg
clave : clave valor : valor estado : estado
freg
tabla = **vector [1 ..TV) de** *triple*
ftipos

Crear la tabla vacía consiste en inicializar todas las posiciones del vector como libres, con un coste en 0(7V).

fun tabla-vacíaO **dev** *t : tabla { Q(N))*
para *i* **= I hasta** *N* **hacer**
f[í).eszoc/o := libre
fpara
ffun

Utilizamos una función auxiliar localizar que busca en la tabla una clave, siguiendo la sucesión de posiciones indicada por la operación prueba. Suponemos que prueba! l.c) = *h(c).* La función localizar devuelve tres resultados. La variable *encontrarlo* indica si se ha encontrado la clave. En caso de encontrarse. la variable *p* indica la posición en la que se encuentra. En cualquier caso, la

variable </ indica la primera posición en la sucesión de saltos que está libre o borrada (si se ha encontrado alguna). Cuando la clave no esté, *q* se utilizará para insertar ahí la clave, y cuando esté, *q* se utilizará para adelantar la clave.

fun localizarle : *clave, t : tabla)* **dev** *(encontrado : bool. p. q : nat)*
encontrado := falso ; *q* := 0 ; *salir* := falso ; *intento :=* **I mientras** -•*salir* A *intento* < *N* **hacer**
/ := prueba(intento, c)
casos
t[i |*.estado* = libre —> (la clave no está)
salir := cierto
si </ = 0 entonces *q := i* fsi
0 r[í *].estado* = ocupada ->
si r[í].c/ave = *c* entonces (la clave sí está)
salir cierto : *encontrado :=* cierto : *p := i*
si no (hay que seguir) *intento :=* *intento* **4- 1** fsi
0 r[r *].estado* = borrada -> (hay que seguir)
si *q* = 0 entonces *q :=* r fsi
intento := intento **+ 1**
fcasos
fmientras
ffun

El coste de localizar depende del tamaño *N* de la tabla, de la *tasa de ocupación* (número de claves diferentes insertadas) y de la función de redispersión utilizada. En el caso peor,

cuando la tabla está llena, el coste está en ©(A'). Sin embargo, si la tasa de ocupación se mantiene baja y la función de dispersión produce pocas colisiones, en la práctica el coste podría ser constante. Para un estudio más protundo del coste de estas tablas, se puede consultar, por ejemplo, el Capítulo 4 de [Fra94|.

Para insertar una asociación entre un valor y una clave, primero se busca la clave en la tabla. Si se encuentra y no se puede adelantar *(q* = 0). la combinación entre el valor en la tabla y el nuevo se almacena en la posición *p.* Si se encuentra y sí se puede adelantar, la información combinada se almacena en la posición *q.* En caso de que la clave no esté, se comprueba si hay posiciones libres (</ = 0). en cuyo caso se inserta en la posición *q.* En otro caso, se produce un error.

proc insertar-tabla(e c : *clave,* **e u :** *valor, t : tabla)*

(encontrado, p.q) := localizar(c, *t)*

si *encontrado* **entonces**

si *q* = 0 **entonces** combinar(t[p).valor, o)

si no { se puede adelantar la clave c)

t[g].c/ave := c; r[^].va/or := t[p].va/or;

combinar(r[<;].*valor, v)*

t[q].eslado := ocupada ;

t[p].estado := borrada

fsi

si no
si *q* = 0 **entonces** error(Espacio insuficiente)
si no
t(q].clave := c; *t[q].valor :*= u;
t[q].estado := ocupada
fsi
fsi
fproc

La implementación de las operaciones está-tabla?, consultar-tabla y eliminar-tabla prácticamente consiste en una llamada a localizar.

fun está-tabla? *(c : clave, t : tabla)* **dev** *b : bool*
(b.p.q) := localizarte, c)
ffun
fun consultar-tabla(c : *clave, t : tabla)* **dev** *v : valor*
(encontrado, p. q) := localizar^, c)
si *encontrado* **entonces** u *:*= *t[p].valor*
si no error(La clave no está)
fsi
ffun
proc eliminar-tabla(e c : *clave, t : tabla)*
(encontrado, p. q) := localizarte, *c)*
si *encontrado* **entonces** *t[p],estado* := borrada **fsi**
fproc

La tabla es vacía si no hay posiciones ocupadas, es decir, todas las posiciones están libres o borradas.

fun es-tabla-vacía?(/ *: tabla)* dev *v : valor { Q(N))*

b •.= cierto ; i := 1
mientras *i* < *N* ᴀ *b* **hacer**
Z» := *(t[i].estado* = libre) v
(t(i],estado = borrada)
i := *i* + 1
fmientras
ffun

Los compiladores de los lenguajes de programación con *estructura de bloques* **usan** *tablas de símbolos* **estructuradas también por bloques. Cuando el compilador está inspeccionando un cierto punto del programa, la tabla de símbolos contiene una sección correspondiente al bloque actual y otras secciones creadas anteriormente, correspondientes a bloques textualmente más externos. En cada sección, la tabla registra las declaraciones de diversos atributos asociados a los identificadores del bloque correspondiente; dichos atributos pueden ser tipos, direcciones de memoria, etc.**

```
program P
var a, b: integer;
   procedure Q(c:integer);
   var a, d: real;
   begin
      writeln(c)
   end
begin
   Q(a)
end.
```

id	atr			
P	prog	—	10230	T_1
a	var	int	1024	
b	var	int	1026	
Q	proc	—	10500	
c	param	int	1028	T_2
a	var	real	1030	
d	var	real	1032	

Figura 7.2: Programa en Pascal y su tabla de símbolos.

Por ejemplo, para el programa en Pascal de la Figura 7.2. la tabla de símbolos mostrada corresponde al momento en que el compilador está inspeccionando la instrucción writeln(c). La tabla se compone de dos secciones: 75. correspondiente al bloque actual, y Ti, correspondiente al bloque textualmente más externo (el del programa principal).

Las dos reglas principales que rigen el funcionamiento de estas tablas de símbolos son: í) cada sección registra como mucho una declaración de atributos para un mismo identiticador: y íí) los atributos asociados a un identificador se consultan comenzando por la sección más recientemente creada de la tabla, correspondiente al bloque actual, y retrocediendo hacia las secciones de bloques más externos.

(a) Especificar el TAD de las tablas de símbolos con estructura de bloques, incluyendo las siguientes operaciones:

o crear la tabla de símbolos vacía (bloque del programa principal).

o crear una nueva sección, entrando en un nuevo bloque.

» insertar en la sección actual la declaración de unos atributos para un

identificador.

o determinar si un identificador está declarado en alguna sección de la tabla.

o determinar si un identificador está declarado en la sección más reciente de la tabla (bloque actual),

o consultar los atributos asociados a un identificador declarado.

® borrar la sección actual, regresando al bloque exterior.

(b) Suponiendo implementados los tipos de los identificadores y de los atributos, definir una estructura de datos adecuada para representar las tablas de símbolos con estructura de bloques e implementar sus operaciones.

------------------**Solución**------------------

Apartado (a)

Las tablas de símbolos están parametrizadas sobre los tipos de los identificadores y las características asociadas a un identificador. El tipo de los identificadores necesitamos que tenga una operación de igualdad, por lo que el parámetro *IDENTIFICADORES* será similar al parámetro *ELEM=* definido en la Sección 1.1.5. aunque con un tipo *ident* en vez de *elemento,* por claridad. Sobre el tipo de las características no existe ningún requisito adicional, por lo que el parámetro *CARACTERÍSTICAS* será como *ELEM* (definido también en la Sección 1.1.5).

especificación *TABLAS-DE-SÍMBOLOS[IDENTIFICADORES,*

CARACTERÍSTICAS!
usa *BOOLEANOS*
tipos *tabla-sim*
operaciones
crear: —>*tab/a-sím* (constructora)
entrar*:* *tabla-sím*—> *tabla-sím*(constructora)
declarar *:* *tabla-símident caract* — *tabla-sím* (constructora)
declarado? *:tabla-sím ident* —> *bool*
declarado-local? *tabla-sím ident* —> *bool*
consultar*:* *tabla-sím ident* —> ,, *caract*
salir *: tabla-sím* —*tabla-sím*
variables
t : tabla-sím
i, j : ident
c.d : caract

Las constructoras son crear, entrar, que sirve para separar los diferentes bloques en la tabla, y declarar. Las constructoras *no son libres,* pues el orden de las declaraciones de identificadores dentro de un mismo bloque no importa, siempre que las declaraciones sean conectas. Además declarar es *parcial* porque no es posible declarar dos veces un mismo identificador en el mismo bloque: consultar solo está definida cuando el identificador existe: y salir solo está definida cuando no se está en el bloque más externo.

ecuaciones

declarar^, *i,* c) = error 4= declarado-local?(r, *i)*

declarar(declarar(í, *i.* c), *j.* r/) = declarar(declarar(r. *j. d). i.* c) <= *i^j*

Las restantes operaciones se definen en términos de las constructoras, distinguiendo los tres casos. Como la constructora declarar es parcial, para simplificar la presentación de la especificación escribimos las ecuaciones suponiendo que todos los términos que involucran declarar están bien definidos. El único detalle destacable es que cuando se crea un nuevo bloque con entrar no hay nada declarado en ese bloque y, por tanto, declarado-local? devuelve falso en ese caso.

declarado?(crear, i) = falso

declarado?(entrar(/), í) = declarado?(r. i)

declarado?(declarar(/, *j. c). i) = i == j* v declarado?(r, í)

declarado-local? (crear, í) = falso

declarado-local?(entrar(r), i) = falso

declarado-local?(declarar(/, *j. c), i) = i == j* v declarado-local?(/. í) consultar^. í) = error <= ~-declarado?(r, i)

consultar(entrar(r), í) = consultará, *i}*

consultar(declarar(r. *i. c). i)* = c

consultar(declarar(z, *j. c). i) =* consultar^, *i) <= i ± j*

salir(crear) = error

salir(entrar(r))= *t*

salir(declarar(r. *i, c))*= salir(r)

(especificación

Apartado (b)

En principio, para implementar las tablas de símbolos parece conveniente utilizar tablas (Ejercicio 7.6) donde los identificadores se utilicen como claves (tienen igualdad), y las características se utilicen como valores asociados a las claves. Sin embargo, con una única tabla se confundirían los identificadores de diferentes bloques, que hay que mantener separados. Como el orden en el que se sale de los bloques es el orden inverso al orden en el que se entra en ellos (L1FO), interesa utilizar una *pila de tablas,* donde cada tabla almacene la información asociada a identificadores de un bloque.

tipos
tabla-sím = pila[tabla[ident, caract]]
rtipos

Así, crear una tabla de símbolos vacía consiste en crear una pila con un único elemento, una tabla vacía correspondiente al bloque más externo.

fun crear!) **dev** *t : tabla-sím* { ©(1))
t := pila-vacíaQ
apilar(tabla-vacía(). /)
ffun

Para declarar un nuevo identificador hay que insertar las características asociadas en la tabla que se encuentra en la cima. Si el identificador ya existía en dicha tabla, se produce un error. El coste depende de la implementación de las tablas que utilicemos. Si utilizamos tablas dispersas con operaciones de coste constante, el coste será lineal con

respecto al número de bloques.

proc declarar!/ : *tabla-sim.* e *i : ident.* **e** *c : caract)*
var *bloque-act : tabla[ident, caract]*
  bloque-act := cima!/)
  si está-tabla[9]!!, *bloque-act*) **entonces**
    error(Identif icador ya declarado en el bloque)
  si no
    insertar-tabla(!. c, *bloque-act)*
    desapilar!/) : apilar!*bloque-act. t)*
  fsi
fproc

Pitra comprobar si un identificador está declarado en alguno de los bloques, se consulta en las tablas que representan cada uno de los bloques, desde la cima hacia el fondo de la pila. Para ello, se copia la tabla de símbolos argumento, la cual se recorre consultando la cima y desapilando sucesivamente. Debido a esta copia el coste será lineal con respecto al número total de identificadores declarados en todos los bloques.

fun declarado?!/ : *tabla-sim. i : ident)*
dev *b : bool*
var /' : *tabla-sim. bloque-act tabla[ident, caract]*
  /' := copiar-pila(r) { copiar la tabla de símbolos }
  b falso
  mientras --es-pila-vacía?(r') A -•*b* **hacer**
    bloque-act := cima!/') : desapilar(f')
    b := está-tabla?(í, *bloque-act)*

fmientras
ffun

Para comprobar si un identificador está declarado en el bloque actual, más interno, solo hay que consultar en la tabla de la cima, con un coste constante.

fun declarado-local?(/ : *tabla-sim. i : ident)* **dev** *b : bool*
var *bloque-act : tabla[ident, caract]*
bloque-act := cima(r)
b := está-tabla?(/, *bloque-act)*
ffun

Al consultar las características de un identificador se sigue la misma idea que para declarado?. Se recorren los bloques (tablas) desde la cuna hacia el fondo, terminando en cuanto se encuentra el identifi- cador buscado, con un coste lineal respecto al número total de identificadores.

fun consultará : *tabla-sím. i : ident)* dev *c : caract*
var *t': tabla-sím. bloque-act: tabla[ident, caract]*
  t' := copiar-pila(r) { copiar la tabla de símbolos)
  b := falso
  mientras ~'es-pila-vacía?(r') A -’ZI **hacer**
    bloque-act := cima(r')
    si está-tabla?(í. *bloque-act)* **entonces**
      b := cierto : *c* := consultar-tabla(i. *bloque-act)*
    si no
      desapilar(r')
    fsi
  fniientras
  si *-•b* **entonces** error(Símbolo no declarado) **fsi ffun**

Finalmente, salir del bloque actual consiste simplemente en desapilar y comprobar que no salgamos del último bloque.

proc salir(r : *tabla-sím)* (0(1))
  desapilar(z)
  si es-pila-vacía?(r) **entonces** error(Salida del bloque más externo)
  fsi
fproc

Capitulo 8

8. COLAS CON PRIORIDAD Y MONTÍCULOS

En el Capítulo 4 hemos tratado las colas "ordinarias" en las que se atiende por riguroso orden de llegada (FIFO) Sin embargo, en la vida cotidiana también existe otra clase de colas donde uno tiene la impresión de que siempre son atendidos los demás aunque hayan llegado después. Se trata de colas como las de los servicios de urgencias, en las cuales se atiende según la urgencia, no según el orden de llegada. La estructura lineal de datos que representa este concepto se conoce como *colas con prioridad,* cuyas principales operaciones son añadir un elemento, saber quién es el primero y atender al primero. Cada elemento tiene una prioridad que determina quién va a ser el primero en ser atendido: para poder hacer esto, hace falta tener un *orden total* sobre las prioridades. El primero en ser atendido puede ser el elemento con menor prioridad (por ejemplo, el cliente que necesita menos tiempo para su atención) o el elemento con mayor prioridad (por ejemplo, el cliente que esté dispuesto a

pagar más por su servicio) según se trate de *colas con prioridad de mínimos* o *de máximos,* respectivamente. En la mayoría de los ejercicios de este capítulo nos referimos al primer caso, si bien también tenemos ocasión de tratar el segundo, que resulta completamente simétrico. Para facilitar la presentación de las propiedades de la estructura de cola con prioridad, los elementos se identifican con su prioridad, de forma que el orden total al que nos acabamos de referir es sobre elementos.

Las colas con prioridad pueden implementarse de diferentes maneras. Si se utilizan listas (sin ordenar), añadir puede hacerse en tiempo constante, pero consultar y eliminar el mínimo tienen coste lineal por la búsqueda. Si la lista está ordenada (de menor a mayor), estas dos operaciones tienen coste constante pues el mínimo aparece al principio de la lista, pero añadir un elemento pasa a tener un coste lineal, debido al coste de localizar el punto de inserción. Una estructura para implementar eficientemente una cola con prioridad es un *montículo (heap* en inglés), a la cual se dedica el resto del capítulo.

Conceptualmente un MONTÍCULO es un árbol binario que cumple propiedades adicionales, como detallamos a continuación, pero antes hacemos notar que la terminología que sigue no es uniforme en los libros de texto sobre estructuras de datos. Un árbol binario de

altura *h* es *completo* cuando todos sus nodos internos tienen dos hijos no vacíos, y todas sus hojas están en el nivel *h.* Un árbol binario de altura *h* es *semicompleto* si o bien es completo o tiene vacantes una serie de posiciones consecutivas del nivel *h* empezando por la derecha, de tal manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo. La Figura 8.1 muestra un árbol binario completo y otro semicompleto. pero no completo, obtenido a partir del completo eliminando los tres nodos más a la derecha en el último nivel.

Un ***montículo de mínimos*** es un árbol binario semicompleto donde el elemento en la raíz es menor que todos los elementos en el hijo izquierdo y en el derecho, y ambos hijos son a su vez montículos de mínimos. Equivalentemente, el elemento en cada nodo es menor que los elementos en las raíces de sus hijos y, por tanto, que todos sus descendientes; así, la raíz del árbol contiene el mínimo de todos los elementos en el árbol. La Figura 8.2 muestra un montículo de mínimos cuyos elementos son números naturales.

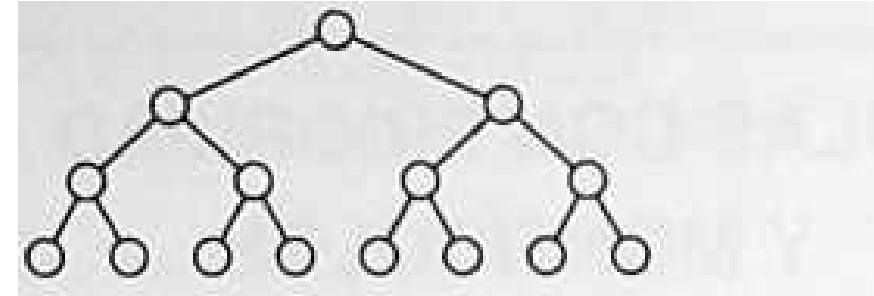




Figura 8.1: Árboles binarios completo y semicompleto.

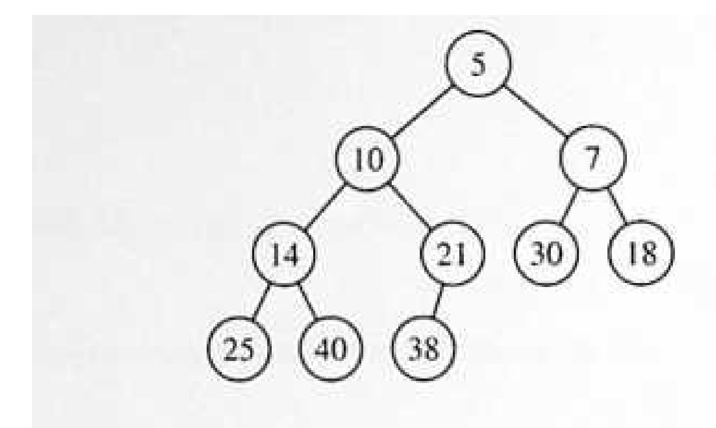


Figura 8.2: Montículo de mínimos.

Un ***montículo de máximos*** se define de manera completamente análoga, exigiendo que cada nodo sea mayor que sus hijos, de forma que la raíz del árbol contiene el máximo de todos los elementos. Por supuesto, según sea de mínimos o máximos, el montículo es apropiado para implementar una cola con prioridad de mínimos o de máximos, respectivamente.

A pesar de que conceptualmente un montículo es un árbol binario, y esta es la mejor I orina de pensar en esta estructura de cara a entender los algoritmos para su manejo, al tratarse de un árbol binario semi- completo, un montículo admite una representación eficiente como vector, que detallaremos en el Ejercicio 8.4. Por tanto, los algoritmos sobre montículos se explican en términos de árboles pero se programan en términos de vectores, es decir, los árboles aparecen de forma conceptual, pero no como estructura de datos explícita que haga uso de las representaciones y operaciones explicadas en el Capítulo 6.

EJERCICIOS RESUELTOS

8.1. Especificar el TAD de las colas con prioridad, con las siguientes operaciones:

- crear una cola con prioridad vacía,
- añadir un elemento,
- consultar el menor elemento,
- eliminar el menor elemento, y
- determinar si la cola con prioridad es vacía.

--------------------**Solución**----------------

Para saber en cada momento cuál es el elemento mínimo, los elementos almacenados en una cola con prioridad tienen que tener definida una relación de orden total. Por tanto, la especificación de las colas con prioridad está parametrizada con respecto al parámetro *ELEM<.* definido en el Ejercicio 5.7. Como constructoras elegimos la operación que crea la cola vacía (cp-vacía) y la que añade un elemento (añadir).

Las operaciones para eliminar y consultar el menor elemento se definen como parciales, estando definidas solo cuando la cola no es vacía.

especificación *COLAS-CON-PFIIORIDAD[ELEM<]*
usa *BOOLEANOS*
tipos *colapr*
operaciones
cp-vacía —> *colapr* {constructora)
añadir *colapr elemento* —>*colapr* { constructora)
mínimo *colapr* —r *elemento*
eliminar-mín : *colapr*—*colapr*
es-cp-vacía?*colapr* —> *bool*

variables

e. f : elemento

cp : colapr

A diferencia de las colas ordinarias sin prioridad (Capítulo 4). ahora las constructoras *no son libres.* ya que el orden en el que se añaden los elementos no importa: el mínimo siempre será el mismo, independientemente del orden de inserción. Las restantes operaciones se definen distinguiendo casos según las constructoras y utilizando la relación de orden para determinar cuál es el mínimo cuando hay más de un elemento

ecuaciones

añadir(añadir(cp, *e). f)* —
añadir(añadir(cp. /). *e)*

mínimo(cp-vacia) = error

mínimo(añadir(cp-vacía, <?)) = *e*

mínimo(añadir(añadir(cp, e),/)) =
mínímo(añadir(cp. e))<=
mínimo(añadir(c/>. e)) < *f*

mínimotañadir(añadir(cp. e),/)) = *f*
<=mínimotañadirfcp. e)) > *f*

eliminar-min(cp-vacía) = error

elimtnar-mín(añadir(cp-vacía. <?)) =
cp-vacía

ehminar-mín(añadir(añadir(<7?, *el. f))*
= añadir(eliminar-
mín(añadir(cp. e)). *f)*
<= minimo(añadir(cp. e)) <
f

eliminar-mín(añadir(añadir(cp. *e). /))*
= añadir(cp.e) <=
mínimo(añadir(cp. e)) > *f*

es-cp-vacia?(cp-vacía) = cierto

es-cp-vacía′(añadir(c7). e)) = falso
fespecificación

8.2. Extender la especificación de los árboles binarios (Ejercicio 6.1) con las siguientes operaciones, teniendo en cuenta las definiciones dadas en la introducción:

- determinar si un árbol binario es completo.
- determinar si un árbol binario es semicompleto, y
- determinar si un árbol binario es un montículo de mínimos.

Solución:

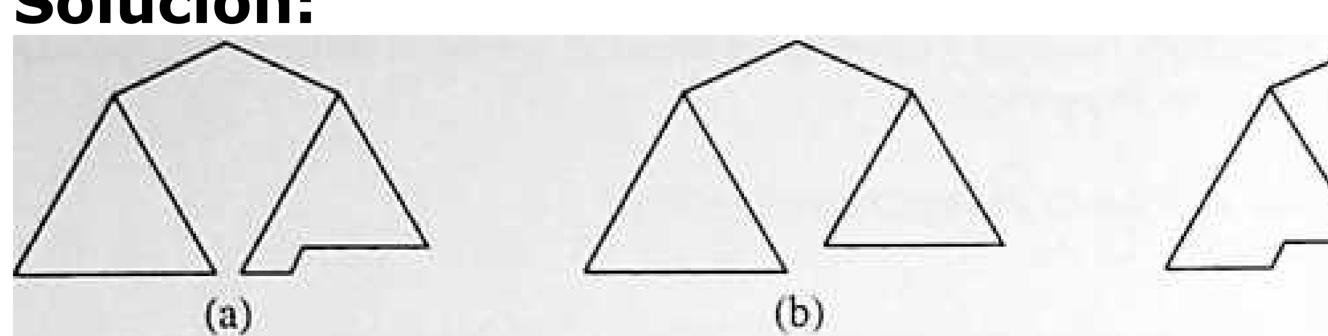


Figura 8.3: Varios árboles semicompletos.

Los conceptos de árbol binario completo y semicompleto están definidos para árboles con elementos de cualquier tipo, por lo que utilizaremos el parámetro *ELEM* para distanciar los árboles binarios. Sin embargo, los montículos solo están definidos sobre elementos con orden, por lo que tendremos que instanciar los árboles binarios con *ELEM<.* En la primera especificación, parametrizada por *ELEM.* definimos las operaciones es-completo? y es-semicompleto?. En la segunda especificación, parametrizada por *ELEM<,* definiremos la operación es-montículo?.

especificación *ÁRBOLES-(SEMI)COMPLETOS[ELEM]*
usa *ÁRBOLES-BINARIOS[ELEMi,*

BOOLEANOS
operaciones
es-completo? *árbol-bin* ■—> *bool*
es-semicompleto? : *árbol-bin* —> *bool*
variables
e : elemento
iz. dr : árbol-bin

El árbol vacío es completo y por tanto semicompleto. Un árbol no vacío es completo cuando al descomponerlo en los dos hijos, ambos son completos y además tienen la misma altura.

ecuaciones
es-completo?(árbol-vacío) = cierto
es-completo?(plantar(íz, *e, dr))* = es-completo?(/z) A es-completo?(r/r) A (altura(iz) == altura(rfr))

Para ver la definición de un árbol semicompleto, consideramos cuántos nodos “faltan” en las posiciones más a la derecha del último nivel. La Figura 8.3 muestra diferentes casos. Si no falta ninguno, el árbol es completo. Si falta alguno, puede ocurrir que todos los que falten sean del hijo derecho, o que falte todo el “último" nivel del derecho y algunos más del hijo izquierdo. En el primer caso (Figura 8.3(a)), el hijo izquierdo es completo mientras que el hijo derecho es semicompleto, y ambos tienen la misma altura. En el segundo caso (Figura 8.3(cj), el hijo izquierdo es semicompleto mientras que el hijo derecho es completo y tiene un nivel menos que el izquierdo, es decir, su altura es una unidad menor Hay un caso

frontera (Figura 8.3(b)), cuando falta todo el “último" nivel del derecho y ningún elemento en el izquierdo; entonces ambos hijos son completos y la altura del derecho es una unidad menor que la del izquierdo. Pero no hace falta considerar este caso aparte porque está incluido en el segundo de los anteriores, pues si un árbol es completo también es semicompleto por definición.

es-semicompleto?(árbol-vacío) = cierto

es-semicompleto?(plantar(íz, *e, dr))* = es-completo?(plantar(íz. *e, dr))* v
(altura(íz) == altura(rfr) A es-completo? (r'z) A es-semicompleto’ír/r)) v (altura(íz) == altura(rfr) + 1 A es-semicompleto?*(iz) A* es-completo?*(dr))*

fespecificación

Ahora definimos la tercera operación haciendo uso de las anteriores.

especificación *ES-MONTÍCULO[ELEM<]*
usa *ÁRBOLES-(SEMI)COMPLETOS[ELEM<], BOOLEANOS*
operaciones
es-montículo? : *árbol-bin —> bool*
operaciones privadas
menor-igual? *elemento árbol-bin —> bool* **variables**
e, f . elemento
iz, dr : árbol-bin

El predicado es-montículo? se especifica directamente según la definición en la introducción: un árbol binario es un montículo de mínimos si es semicompleto, la raíz es menor que todos los elementos en el hijo izquierdo

y en el derecho, y ambos hijos son a su vez montículos de mínimos. Utilizamos la operación privada menor-igual? para determinar si un elemento es menor o igual que todos los elementos en un árbol binario.

es-monticulo?(árbol-vacío) = cierto

es-montículo?(plantar(í-. c. </r)) = es-semicompleto?(plantar(rz, *e, dr))* A
menor-igual?(<?,;-) A
menor-igual?(e. *dr)* A es-
montículo?(í;) A es-
montículo?(<7r)

menor-igual?(e. árbol-vacío) = cierto

menor-igual?(e, plantarfíz, *f. dr)) = e<f* A menor-igual?(e. *iz)* A menor-igual?(e. *dr)* **(especificación**

Demostrar las siguientes afirmaciones:

(a) Un árbol binario completo de altura *h* > 1 tiene 2'~ nodos **en** el **nivel** *i.* para **todo** *i* **entre 1 y** *h.*

(b) Un árbol binario completo de altura *h* > 1 tiene $2^{/1-1}$ hojas.

(c) Un árbol binario completo de altura /; > 0 tiene 2^{A} — 1 nodos.

(d) La altura de un árbol binario *semicompleto* formado por/i nodos es [logrij +1.

------------------**Solución**------------------

Apartado (a)

Se demuestra por inducción sobre el número de nivel *i.*

Cuando í = 1, en el primer nivel solamente hay un nodo que es la raíz, y 2^{1-1} = 1.

Suponiendo el resultado cierto para *i* <

h, como cada nodo en el nivel *i* tiene exactamente dos hijos no vacíos, el número de nodos en el nivel *i* + 1 es igual a 2 * $2'^{–l}$ = 2' = $2^{(, + l)}$->,

Apartado (b)---------------

Las hojas son los nodos en el último nivel *h.* por lo que basta aplicar el resultado del Apartado (a) para *i* = *Ir*

Apartado (c)------------------

Si *h* = 0, el árbol es vacío y el número de nodos es igual a 0 = 2° - 1.

Si *h* > 0, utilizando el resultado del Apanado (a), el número total de nodos es:

h h-1

$2'^{-1}$ = 2^J = 2^A - 1

i=l *j=o*

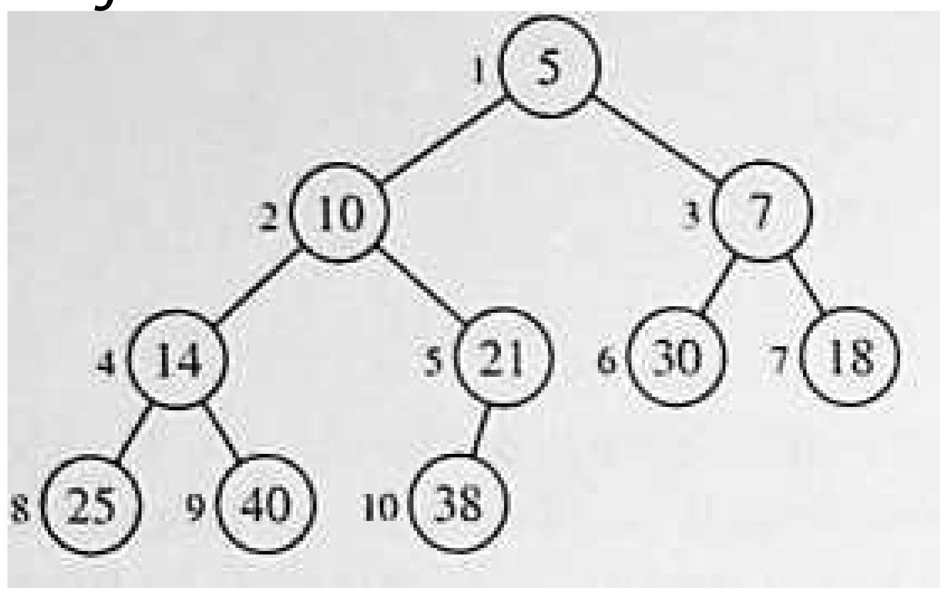


Figura 8.4: Vector representando un montículo de mínimos.

Apartado (d)---------------

Supongamos un árbol binario semicompleto con *n* nodos y altura *h.*

En el caso en que faltan más nodos en el último nivel, o sea, cuando el árbol tiene el número mínimo de nodos (habiendo fijado la altura), entonces es un árbol

binario completo de *h* - 1 niveles más un nodo en el nivel *h,* por lo que. usando el Apartado (c), hay en total $2^{/l-1}$ — 1 + 1 = 2^{h}~^{1} nodos.

En el caso en que el último nivel está todo lleno, es decir, no falta ningún nodo en ese nivel y el árbol tiene el número máximo de nodos (habiendo fijado el número de niveles), tendremos un árbol binario completo de *h* niveles con 2" — 1 nodos, por el Apartado (c).

Resumiendo, tenemos con respecto a *n* la siguiente desigualdad:

$2^{/,-l}$ < *n* < 2^{A} - 1.

Tomando logaritmos en base 2

log($2^{/,-l}$) < log/i < log(2^{A} - 1) < log(2^{A});

equivalentemente.

h — 1 < log *n* < *h.*

es decir, *h* — 1 = [log/tj *y* de aquí *h* = [logítj + 1.

Implementar la estructura de *montículo de mínimos* utilizando un vector, con las operaciones de las colas de prioridad especificadas en el Ejercicio 8.1.

--------------------**Solución**---------------

Un árbol binario se puede representar fácilmente utilizando un vector V[l../V|, para *N* suficientemente grande, mediante la siguiente asignación de posiciones a nodos. La raíz del árbol se almacena en la primera posición del vector, y un nodo almacenado en la posición *i* tiene a su hijo izquierdo en la posición 2i y a su hijo derecho en la posición 2í + 1. Como estas operaciones son ambas inyectivas y devuelven resultados en conjuntos

1	2	3	4	5	6	7	8	9	10
5	10	7	14	21	30	18	25	40	38

disjuntos (números pares e impares, respectivamente), cada nodo de! árbol se almacena en una posición diferente del vector. Con esta representación también es fácil conocer al padre de un nodo, pues el padre del nodo almacenado en la posición *i* es el que se encuentra en la posición *i* div 2.

En general, esta representación puede dejar muchos “huecos" en el vector; sin embargo, cuando el árbol binario es *semicompleto,* como es el caso de un montículo, esta representación corresponde al recorrido del árbol por niveles y no deja huecos en el vector entre las posiciones 1 *y N,* donde *N* es ahora el número de nodos de! árbol, como se ilustra en la Figura 8.4.

Según la numeración anterior, el primer nodo del nivel *j* se coloca en la posición *l'*~^{1}. Por otra parte, usando el Apartado (a) del Ejercicio 8.3, el número total de nodos en los niveles del 1 al *j* — 1 es igual a z-1

$2^{<:-1} = 2^{J_l}$ — i

k=l

Así. no puede haber huecos hasta la posición $2\text{-}^{'-1}$. Además, los elementos del nivel *j* del árbol se encuentran de forma consecutiva entre las posiciones $2^{'-1}$ y 2^{J} — 1.

Una vez explicada la representación de un árbol semicompleto en un vector, pasamos a realizar la implementación de

los montículos.

El tamaño del vector *N* está fijo e indica el número máximo de elementos del montículo. Utilizamos también un número natural *último* para indicar el número de elementos del montículo en un momento dado (equivalentemente, la última posición ocupada en el vector). Así. el elemento de la posición *i* tiene hijo izquierdo si 2/ < *último* y tiene hijo derecho si 2/ + 1 < *último.*

tipos

montículo **= reg**

V[1 ../V] de *elemento*

último : 0..N

freg

ftipos

Nótese que el requisito de orden entre los elementos que se debe cumplir en un montículo no se refleja en el tipo, por lo que este realmente representa árboles semicompletos. El requisito de orden se logrará por cómo se añaden y eliminan elementos.

El montículo vacío no tiene elementos.

fun montículo-vacíoO dev *M : montículo* (0(1))

M .último **:= 0 ffun**

Solamente hay una forma de añadir un nodo a un árbol semicompleto manteniendo esta propiedad: si “faltan" nodos en el último nivel, el nuevo nodo será el siguiente (en el recorrido por niveles) al último nodo de ese nivel, y si el árbol de partida es completo el nuevo nodo será el de más a la izquierda en un

nuevo nivel. En la representación como vector de un árbol semicompleto esto se refleja en que el nuevo elemento se coloca en la posición siguiente al *último.* Pero este elemento no tiene por qué estar bien colocado en esa posición con respecto a la propiedad de orden en el montículo; para mantener dicha propiedad, el elemento tiene que "flotar", mediante la operación auxiliar flotar que veremos a continuación. El coste de añadir es proporcional al de flotar que como veremos es logarítmico respecto al número de elementos en el montículo.

proc añadir (A-/ : *monticulo.ee : elemento)* **(0(log** *M. último))*
 si *M.último = IV* **entonces**
error(Espacio insuficiente)
 si no
  M.último := M.último + 1
  M.V[M.último] := e
  flotar(A7. V, *M.último)*
 fsi
fproc

El elemento a flotar no está bien colocado si es menor que su padre, por lo que hay que intercambiarlo con este. El proceso se repite hasta que el padre sea menor o igual, o lleguemos a la raíz. El coste es proporcional a la altura del montículo porque en cada iteración del bucle se sube un nivel en el árbol. y como se demuestra en el Ejercicio 8.3 la altura es logarítmica respecto al número de elementos en el **montículo.**

proc flotadV[l..TV] de elemento, e *j* :

l.TV) { 0(logj))
 í := *j*
 mientras *i* / 1 Ac V[í] < V[í div 2] hacer
  (V[í], V[í div 2]) := (V [z div 2], *V[i])*
  i := *i* div 2
 fniientras
fproc

En un montículo, el mínimo se encuentra en la raíz, o sea, en la primera posición del vector.

fun mínimo] A7 : *montículo')* **dev** *e : elemento* { 0(1))
 si *M.último* = 0 **entonces** error(Montículo vacío)
 si no *e* := Af.Vjl]
 fsi
ffun

Para eliminar el mínimo hay que quitar el elemento en la primera posición del vector y seguir manteniendo un montículo en el vector. Para conseguir que siga siendo semicompleto, el hueco que aparece en la primera posición del vector al borrar el mínimo se cubre con el elemento en la última posición *(último).* Para mantener el orden correcto entre los elementos, el elemento movido nene que "hundirse", lo que hace la operación auxiliar hundir. El coste de eliminar-mín es proporcional al de hundir que también es logarítmico respecto al número de elementos en el montículo.

proc eliminar-mín(M : *montículo)* (0(log *M.último))*
 si *M.último* = 0 **entonces** error(Montículo vacío)

si no
/W.V[1] := *M.V[M.último]*
M.último := M.último — 1
hundid AT. *V.* 1. *M.último)*
fsi
fproc

El elemento a hundir no está bien colocado si es mayor que alguno de sus hijos. El problema se resuelve intercambiando el elemento con el menor de los hijos, y repitiendo el proceso hasta que el elemento que se hunde sea menor o igual que sus hijos, o lleguemos a una hoja. El coste es proporcional a la altura del montículo porque en cada iteración del bucle se baja un nivel en el árbol, y como sabemos la altura es logarítmica con respecto al número de elementos en el montículo (véase el Ejercicio 8.3).

{><<')
proc hundid V[1..TV] **de** *elemento,* **e** *j, k* : 1..TV) (0(log¿))
fin := falso ; *i := j*
mientras 2 * *i* < *k* /\ -•*fin* **hacer** (la posición *i* no es una hoja)
{ mínimo de los hijos |
si 2 * *i* **+ 1 <** *k* Ar V [2 * *i* 4- 1] < V [2 * *i* **]** **entonces** *m* := 2 * *i* + I
si no *m* := 2 * *i*
fsi
{ si es necesario, se intercambia con el mínimo de los hijos)
si V[m] < V[í] **entonces** (V[Z], V[m]) := (V[„r], V[í]) ; *i := m*
si no *fin :*= cierto
fsi

fniientras
fproc

Un montículo es vacío cuando no tiene elementos.

fun es-montículo-vacío?*(M : montículo)*
dev *b : bool* **{ 0(1))** *b* := *(M.último* = 0)
ffun

8.5. (a) Implementar dos versiones de un procedimiento monticulizar que transforma un vector **dado en un** vector que implementa la estructura de un montículo. La primera versión utiliza flotar mientras que la segunda utiliza hundir (véase la solución del Ejercicio 8.4). Dado un mismo vector inicial, ¿generan las dos versiones el mismo montículo?

(b) Demostrar por inducción que

k
52j2-' = (A--l)2`+l+2.
;=I

Deducir que el coste en el caso peor de la creación de un montículo utilizando flotar está en <r)(7V log *N).*

(c) Demostrar que

y'i. = o-lL±l Z—v 7J " *->k* ■
J = I "

Deducir que el coste en el caso peor de la creación de un montículo utilizando hundir está **en 0(.V).**

-----------------Solución------------------

Apartado (a)

La primera versión recorre el vector de izquierda a derecha, y mantiene que las posiciones por las que ya ha pasado forman un montículo, representado en dichas posiciones del vector. Cuando se

avanza a la siguiente posición, se aplica la misma idea que para añadir un elemento a un montículo: el nuevo elemento se incorpora a los que forman el montículo haciéndolo flotar entre estos hasta que ocupe el lugar apropiado.

proc monticulizarl (I^z| I.. A' | de *elemento)*

para *j* = 2 **hasta** *N* **hacer**

flotare V. *j)*

fpara

fproc

El procedimiento anterior se basa en iterar la operación flotar. También es posible formar el montículo iterando la operación hundir. La idea es que cualquier vector se puede ver como la representación de un árbol semicompleto, como ya se ha explicado al principio de la solución del Ejercicio 8.4. Para obtener a partir de él un montículo tenemos que reorganizar los elementos de forma que los descendientes de cada uno sean mayores o iguales. Esto se consigue por niveles empezando por las hojas. Cada hoja por separado constituye automáticamente un montículo. A continuación se forman montículos más grandes de dos o tres nodos, colocando encima de cada par de hojas el nodo correspondiente del nivel superior, su padre en el árbol; para colocar correctamente el elemento hay que "hundirlo". Se procede de esta manera, intuitivamente subiendo por niveles, hasta llegar a la raíz de todo el árbol.

Nótese que. como los elementos del árbol semicompleto están colocados en su representación como vector siguiendo un recorrido por niveles, para recorrer los niveles desde las hojas hasta la raíz se puede hacer un recorrido de derecha a izquierda en el vector y. como las hojas de un árbol semicompleto de tamaño A’ se encuentran a partir de la posición *N* div 2+ I en el vector, basta hacer ese recorrido empezando en la posición A' div 2.

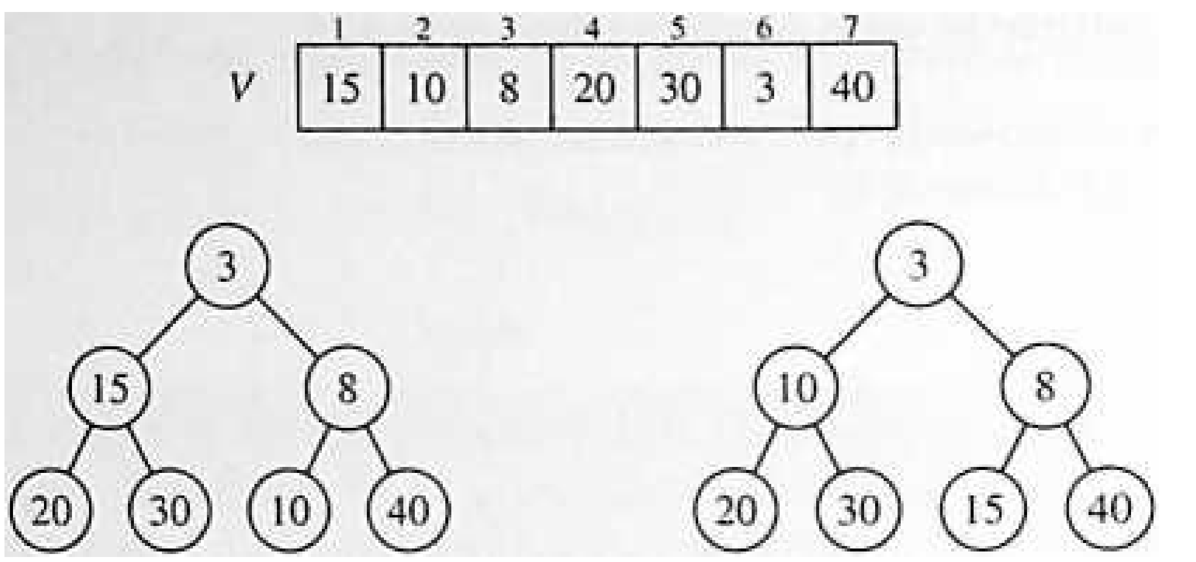


Figura 8.5: Dos montículos obtenidos a partir del mismo vector.

proc monticulizar2(V[l..N] **de** *elemento)*
para *j* = *N* div 2 **hasta** l **paso** — l **hacer**
hundir(V, *j, N)*
fpara
fproc

El montículo obtenido utilizando uno u otro método no tiene por qué ser el mismo, como muestra la Figura 8.5. El montículo de la izquierda se obtiene a partir del vector *V* utilizando monticiilizarl, mientras que el de la

derecha se obtiene utilizando monticulizar2.

Apartado (b)----------------------

Como caso básico consideramos k = 1; entonces la suma vale 2. mientras que la expresión de la derecha vale (1 — l)2^{1+l} +2 = 2.

Suponiendo como hipótesis de inducción la propiedad cierta para k — 1. consideramos el caso k:

ÍP-J =

;=i ;=i

=*•'•(k - 2)2^k + 2 + $k2^k$

=(2k - 2)2* + 2

=(A-- 1)2*+1+2.

Cuando se crea el montículo utilizando flotar, vamos de arriba hacia abajo (en un árbol binario semi- completo). En el segundo nivel hay 2 (= 2^{2-1}) elementos que “flotan" cada uno 1 (= 2 - 1) vez en el caso peor; en el tercer nivel hay 4 (= 2^{3-1}) elementos que "flotan” cada uno 2 (= 3 - 1) veces, etc. En general, en el nivel i hay 2'"1 elementos que “flotan" cada uno i — 1 veces en el caso peor. Y así sigue el proceso hasta el último nivel h = [log N j + 1 (por el Ejercicio 8.3), que en el caso peor está lleno y tiene 2/,–l elementos que “flotan” cada uno h — 1 veces. Nótese que cada vez que se baja un nivel en el árbol binario (completo), hay el doble de elementos que en el anterior y cada uno de ellos "flota" una vez más. Por tanto, el coste en el caso peor es

/i A-l

E('-D2'-’ =

i=2 j = l

= (/i - 2)2* + 2

= (LlogAJ - l)2tlogA,J+l+2

e e>(Llog WJ2UosNJ)

= ©(/VlogN).

Apartado (c)-------------------

La siguiente demostración *no* es por inducción, sino por manipulación algebraica, usando el primer resultado del apartado anterior.

k . k

Ei = ¿E^

7 = l " j=l

l k-'

= ^Eí*-»2'’

1=0

, *-l . A-l

=

í=0 í=0

= A(2‘-l)-l((¿-2)2* + 2)

k 2

= *k-^-(k-2)-^*

- 2-^E

2*

Cuando se crea el montículo utilizando hundir, vamos de abajo hacia arriba (en un árbol binario semicompleto). Todos los elementos que están en el último nivel *h* no se mueven. Los elementos en el penúltimo nivel *h* — 1 "se hunden" cada uno 1 vez en el caso peor; todos los del antepenúltimo nivel *h — 2* “se hunden" cada uno 2 veces en el caso peor, etc. Al final, el elemento en la raíz “se hunde" *h*

— 1 veces. En general, en el nivel *i* hay 2[1]"[1] elementos que “se hunden" cada uno *h* — *i* veces en el caso peor. Nótese que ahora cada vez que se sube un nivel en el árbol binario (completo), hay la mitad de elementos, aunque cada uno de ellos “se hunde" una vez más. Por tanto, el coste en el caso peores

A-l

- r)2'$^{—1}$

/=1

$$< \sum_{j=1}^{h} j2^{h-j}$$

$$= 2^h \sum_{j=1}^{h} \frac{j}{2^j}$$

$$= 2^h(2 - \frac{k+2}{2^k})$$

$$\leq 2^{h+1}$$

$_2$llogiVJ+2

De aquí se obtiene que el coste en el caso peor está en *O(N).* y como obviamente está en Q(A') porque se recorre la mitad del vector, se concluye que está asimismo en 0(/V).

8.6. El algoritmo de ordenación por el *método del montículo* (heapsort) inserta en un montículo todos los elementos del vector a ordenar. Después se va extrayendo sucesivamente el mínimo del montículo, por lo que los elementos salen en orden creciente.

(a) Implementar este algoritmo de ordenación utilizando una cola de prioridad de torma abstracta.

(b) Impleinentarlo utilizando el mismo

vector para representar el montículo.

------------------Solución-------------------

Apartado (a)

Como se explica en el enunciado, el algoritmo de ordenación por el método del montículo consta de dos fases. En la primera, todos los elementos del vector a ordenar se añaden uno a uno a un montículo inicialmente vacío. En la segunda fase, se va extrayendo sucesivamente el mínimo del montículo y se va colocando en la correspondiente posición del vector, que se recorre de izquierda a derecha.

proc heapsort-abstracto(V[l..lV] **de** *elemento)* (0(/V logA')) **var** *C : colapr*
C := cp-vacía()
para *i* = 1 **hasta** *N* **hacer**
añadir(C, V[í])
fpara
para *i* = 1 **hasta** *N* **hacer**
V[r] := mínimo(C)
eliminar-min(C)
fpara
fproc

El coste del algoritmo depende de la implementación concreta de la cola de prioridad. Supongamos que se utiliza un montículo. En la primera fase, los elementos se van añadiendo a un montículo cuyo tamaño va aumentando de 0 a *N.* El proceso es el mismo que en el procedimiento monticulizarl del Ejercicio 8.5. por lo que su coste está en 0(W log *N).* En la segunda fase, se consulta y extrae sucesivamente el

mínimo de un montículo cuyo tamaño varía de *N* a 0. Sabiendo que el coste de eliminar el mínimo en un montículo de tamaño *j* está en 0(log j). concluimos que el coste de la segunda fase está también en 0(/V log *N)* (véase el Ejercicio 7.3). Por tanto, el coste de heapsort-abstracto está en 0(7V log *N).* El espacio adicional utilizado está en *&(N),* por el tamaño del montículo auxiliar.

Apartado (b)---------------------

Podemos ahorramos este espacio adicional si utilizamos el mismo vector para representar el montículo auxiliar. Primero el vector se convierte en un montículo utilizando el procedimiento monticulizar2 del Ejercicio 8.5. Después se recorren las posiciones del vector de derecha a izquierda En todo momento el vector se divide en dos partes: la de la izquierda representa el montículo del cual vamos extrayendo cada vez el mínimo para colocarlo al principio de la parte de la derecha, la cual contiene los elementos extraídos ordenados de mayor a menor.

mon	orde

El tamaño del montículo va decreciendo mientras que la parte ordenada va creciendo hasta obtener un vector totalmente ordenado.

proc idea-heapsort(V[1 ../V] **de** *elemento)* { 0(7V log *N))*
monticulizai^fV)

para *j* = *N* **hasta** 2 **paso** - 1 **hacer**
(V[1J. V[j]> := <V[J], V[l])
hundirfV, 1. *j* — 1)
fpara
fproc

El resultado de este algoritmo es el vector ordenado *de mayor a menor.* Para obtener el vector ordenado de la forma habitual, de menor a mayor, y continuar haciendo todas las operaciones sobre el mismo vector sin ayuda de ningún vector auxiliar, hace falta un *montículo de máximos* en vez del montículo de mínimos. Para lo que nos interesa en este ejercicio, a saber, implementar el algoritmo de ordenación por el método del montículo, lo único que hay que cambiares la operación de comparación entre elementos (pero no entre índices) dos veces en el procedimiento hundir, que pasa de < a >. Para la implementación de un montículo de máximos habría que adaptar de la misma forma el procedimiento flotar, pero de hecho no hace falta cambiar las demás operaciones que hemos visto en el Ejercicio 8.4.

(;<<-)
proc hundir-máx(V| l..jV] **de** *elemento,* **c** *J.k* : 1../V) j 0(logit))
fin := falso ; í *:= j*
mientras 2 * *i*< *k* A -'*fin* **hacer**
(máximo de los hijos)
si 2 * *i* 4- I < *k* Af *V* [2 * *i* + I] > *V* [2 * *i*] **entonces** *m* := 2 * *i* + l

si no *ni := 2 * i*
fsi
(si es necesario, se intercambia con el máximo de los hijos |
si V[m] > V[í] entonces (V[í], V[m]) := (V[m], V[í]) ; *i := m*
si no *fin* := cierto
fsi
fmientras
fproc
proc heapsorti V| l../V] de *elemento)* (0(*N* log *N))*
(monticulizar j
para *j* = *N* div 2 **hasta** l **paso** - l **hacer**
hundir-máx(V. *j. N)*
fpara
para *j* = *N* **hasta** 2 **paso — I hacer**
(VIH. V|./|> := V[l]>
hundir-máx(V. I. j — l)
fpara
fproc

Implementar una estructura de datos que soporte las siguientes operaciones con el coste pedido:

- crear una estructura vacía, con coste constante.
- consultar el máximo de todos los elementos, con coste constante.
- consultar el mínimo de todos los elementos, con coste constante.
- borrar el máximo, con un coste en O(log *N).*
- borrar el mínimo, con un coste en O (log *N).* e
- insertar un elemento, con un coste en O(IogiV).

donde *N* es el número de elementos en la estructura sobre la cual tiene lugar la acción.

-------------------Solución------------------

Según la solución del Ejercicio 8.4. un montículo de mínimos permite consultar el mínimo con coste constante, borrarlo con coste logarítmico, e insertar un elemento también con coste logarítmico. Sin embargo, para calcular el máximo habría que buscar entre todas las hojas, con un coste lineal; el coste de eliminarlo sería semejante. Si utilizamos en cambio un montículo de máximos, la situación se invierte. Para conseguir el coste eficiente en lodos los casos, como se pide en el enunciado, hay que trabajar de alguna forma simultánea con un montículo de mínimos y otro de máximos. Por tanto, para representar la estructura, utilizaremos dos vectores *VM y Vin* que

corresponden a los montículos de máximos y mínimos, respectivamente. Con esto se logrará que las operaciones tengan el coste deseado. Sin embargo, cuando un elemento se elimina de un montículo también tiene que ser eliminado del otro. Para hacer esto de forma eficiente, los elementos almacenados en los vectores *VM* y *Vm* son pares de forma que cada dato lleva consigo la posición que ocupa en el otro montículo. Cuando se modifica uno de los dos montículos (esencialmente, al hundir o flotar), se tiene que actualizar adecuadamente la información de posiciones en el otro vector. El diagrama de la Figura 8.6 ilustra la idea, representando esquemáticamente cada montículo como un árbol. Si en *VM* aparece la información (*e, j*) en la posición p. entonces en *Vm* aparece la información (*e. p*) en la posición *j.* y viceversa.

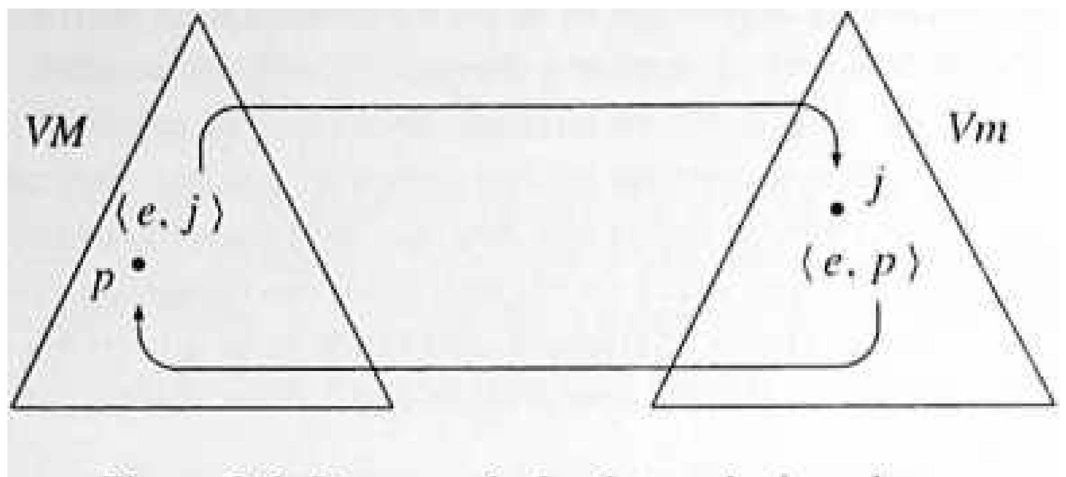


Figura 8.6: Dos montículos interrelacionados.

```
tipos
    elem-pos            reg
                            elem : elemento
                            pos : \..N
                        freg
    montículo-doble     reg
                            VM[1 ..N] de elem-
                        pos
                            V7n[l../V] de elem-
                            pos último : 0..N
ftipos
```

La estructura vacía no tiene elementos en ninguno de los montículos.

fun crear-mdO **dev** *D : montículo-doble* (0(1))
D.último := 0
ffun

El máximo se encuentra en la primera posición de *VM y* el mínimo en la primera posición de *Vm.*

fun máximo-md(D : *montículo-doble)*
dev *e : elemento {©(!)}* **si** *D.último =* 0 **entonces** error(Montículo vacío) **sino** *e := D.VM[l].elem* **fsi**
ffun

fun mínimo-md(£> : *montículo-doble)*
dev *e : elemento* { 0(1)) **si** *D.último =* 0 **entonces** error(Montículo vacío) **sino** *e := D.Vm[l].elem* **fsi**
ffun

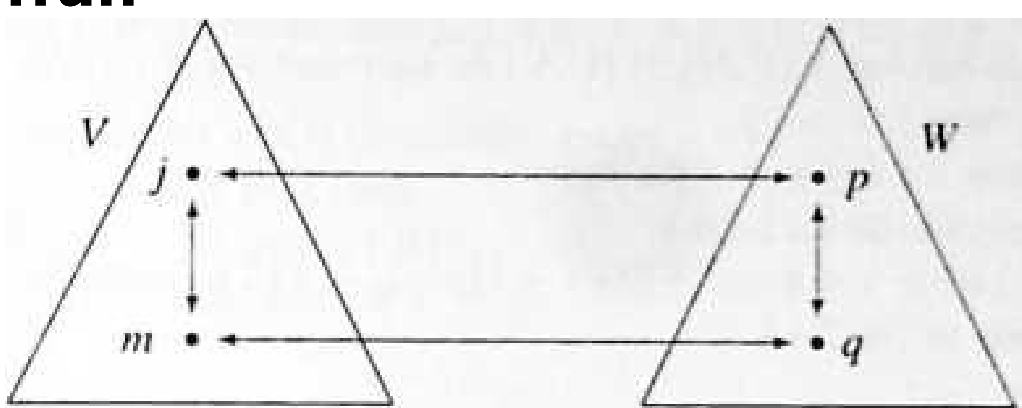


Figura 8.7: Esquema de las variables para hundir y flotar.

Las operaciones de eliminación siguientes se basan en quitar la raíz del montículo (visto como árbol), es decir, el elemento en la posición 1. que denominaremos (*e. p*). y poner en su lugar el elemento en la última posición *último.* Luego se utiliza la operación auxiliar de hundir para que el elemento de la raíz se coloque en su lugar. La

complicación adicional respecto al Ejercicio 8.4 se debe a que al quitar o mover un elemento en un vector, hay que hacerlo también en el otro. En el otro vector se quita el elemento de la posición />. se pone en su lugar el elemento en la última posición *último,* y se flota para ponerlo en su lugar. Obsérvese que el elemento movido solamente puede flotar, porque se ha colocado en la posición donde había un máximo (mínimo), por lo que. si tiene descendientes, solo pueden ser iguales al máximo (mínimo). En el caso especial en que *p = último* no hay que hacer este último paso porque se elimina el elemento en la última posición.

proc eliminar-máx-md(*D : montículo-doble)* (O(log *D.último))*
 si *D.último* — **0 entonces**
 error(Montículo vacío) **si no**
  p := D. IW(1 *].pos ; q := D.VM] D.último].pos*
  D.VA7[1] *= D.VM[D.último] ; D.***Vni[r/].pos := 1**
  si *p /) último* **entonces**
   i := D.Vm[D.último].pos ; D.V'nijp] := *D.Vm]D.último] : D.VM]r].pos := p*
   flotar-mín-md(*D. Vm. D.VM. p)*
  fsi
  D.último := D.último — 1
  hundir-máx-md(*D.VM. D.Vm.* 1. *D último)*
 fsi fproc

proc eliminar-mín-md(Z) : *montículo-doble)* (9(log *D.último)* |
 si *D.ultimo* = **0 entonces**

error(Montículo vacío) **si no**
p := D.Vm] 1)./w.s ; *q := D.Vm]D último].pos*
D.Vm[l] := *D.Vm]D.último] ; D.VM[q].pos* := 1
si *p D último* **entonces**
r := D VM]D.último].pos ; D.VM[p] := D.VM]D.último] ■. *D.Vm[r].pos := p*
flotar-máx-md(*D.VM, D Vm. p)*
fsi
D.último := D.último — 1
hundir-min-md(£).V'»i. *D.VM.* 1. *D.último)*
fsi
fproc

El diagrama de la Figura 8.7 ilustra las relaciones entre las principales variables que aparecen en los algoritmos hundir-mín-md y hundir-máx-md que se describen a continuación. El mismo diagrama sirve también para los algoritmos flotar-mín-md y flotar-máx-md poniendo *j* div 2 en lugar de *m.*

(j < M
proc hundir-mín-md(V[l ..AT], W[1..N] **de** *elem-pos. e j. k* : 1..7V) | 0(logA'))
fin := falso ; *i := j* **mientras** 2 * *i* < *k* /\ -*fin* **hacer**
{ mínimo de los hijos)
si 2 * *i* + 1 < *k* /\c V [2 * *i* + Ij.e/eni < V [2 * *i].elem* **entonces** *ni* := 2 * *i* + 1
si no *ni := 2* i* **fsi**
(si es necesario, se intercambia con el mínimo de los hijos)

(y se actualizan las posiciones en el otro vector (**si V[m].e/c/n < V[í].e/em entonces**

p := V[í].po.r ; *q* := Vpnj.pos

<V[í],V[m]) := <V[m],V[í])

lV[p].pos := *ni ;* lV[<7].pos := *i i :=* m

si no *fin* := cierto

fsi fmientras fproc *{ j* < * 1 **proc** hundir-máx-md(V[l..yV], fV[l../V] **de** *elem-pos.* **c** *j. k* : 1../V) (0(logA) }

fin := falso : *i := j* **mientras** 2 * *i* < *k* A *-fin* **hacer**

{ máximo de los hijos)

si 2 * *i* + 1 < *k* Ac V[2 * *i* 4- Ij.e/em > *V[2 * i].elem* **entonces** *ni* := 2 * *i* + I

si no *m* := 2 * *i* **fsi**

(si es necesario, se intercambia con el máximo de los hijos)

{ y se actualizan las posiciones en el otro vector)

si *V[m].elem* **> V[í].e/em entonces**

p i— VjíJ.pos ; *q* := VfmJ.pos

<V[Í], V[m]) := (V[rn],V[í])

IV[p].po5 := *ni ;* IVjr/j./xw := *i i :=* *ni* **si no** *fin :=* cierto **fsi fmientras fproc**

Para añadir un elemento a la estructura, hay que añadirlo a ambos montículos en la posición siguiente a la última ocupada, y en ambos montículos el elemento tiene que "flotar".

proc añadir-md(Z) : *montículo-doble,* **e** *e : elemento)* { 0(log *D.último)* | **si** *D.último = N* **entonces** error(Espacio insuficiente) **si no** *D.último := último +*

I

D.Vni[D.último].elein := e

D.Vm[D.último].pos := D.último

D.VM[D .último].elein := e

D.VM[D.últiino].pos := D.último

doble-flotar-mín(D.V/„i. *D.VM. D.último)*

doble-flotar-máx(£>.V'M, *D.Vm. D.último)* **fsi**

fproc

```
proc flotar-mín-md(V'| L.A'], W[ 1 ..Ai]
de elem-pos.ek : 1../V) ( O(logí) i
 i := k
 mientras í / 1 Ar V[i].elem < V[i div
2].elem hacer
   /> := V[/].po5 ; <7 := lz[í div 2].pos
   ( V[í], V[í div2]) := <V[ídiv2], V[i]>
   WI/•>] .pos '■= i div 2 ; Wjz/J.pos := i
   i := i div 2
 imientras
fproc
proc flotar-máx-md( V| I..AT], W[ 1 ,.AÍ]
de elem-pos. e k : 1..AÍ){ 0(log Jt) }
 i := k
 mientras i I Ar V[i].elem > V[í div
2],elem hacer
   p := V[/].po5 ; q := V(í div 2].pos
   ( V[z ], V[7 div 2] > := ( V[í div 2],
V[í])
   WjpJ.pos := í div 2 ; W[<7].po5 := i
   i ■= i div 2
 (mientras
fproc
```

Las ideas para la solución desarrollada en este ejercicio están tomadas de un ejercicio del libro de Parberry |Par95], pero existen otras estructuras basadas en montículos para resolver el mismo problema: entre ellas podemos destacar los *montículos min-max* y los *deaps* que se tratan en el Capítulo 9 del libro de Horowitz y Sahni [HS94].

Supongamos que queremos utilizar un montículo para almacenar pares de la forma (*elem. prioridad}* donde *elem* es un número natural en el intervalo 1..AL las prioridades son números reales, y el *elem* de todos los pares es diferente. El orden entre los pares viene inducido por el orden entre las prioridades. Implementar el montículo de tal forma que se pueda modificar la prioridad asociada a un elemento en el montículo y mantener las propiedades de la estructura en tiempo logarítmico.

-----------------**Solución**------------------

El montículo se representa en un vector \$^z$[1..*N]* como en el Ejercicio 8.4. aunque ahora los elementos serán pares (*elem. prioridad}.* Para poder saber en tiempo constante dónde está en el montículo un *elem* concreto, utilizaremos además un vector *posiciones]* I..A'] tal que *posiciones]!*] es la posición de V donde se encuentra el elemento *i. V]posiciones]i]].elem* = *i.* Si el elemento *i* no está en el montículo. *posiciones]!]* = 0. El tipo representante es el siguiente:

tipos

par **= reg**

elem : 1.. N

prioridad : real

freg

montículo-pares = **reg**

I^z[1..A'] de *par*

posiciones]l..N] **de 0..A'**

último : 0..N

freg

ftipos

La unplementación de las operaciones es muy similar a la vista en el Ejercicio 8.4 para montículos simples, solo que ahora cualquier modificación de V tiene que reflejarse también en *posiciones.*

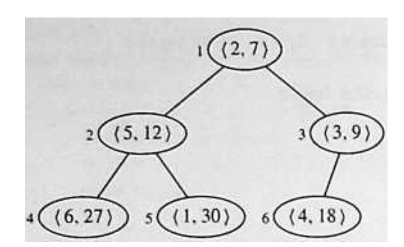


Figura 8.8: Montículo de pares y su representación con dos vectores.

El montículo vacío no tiene ningún elemento.

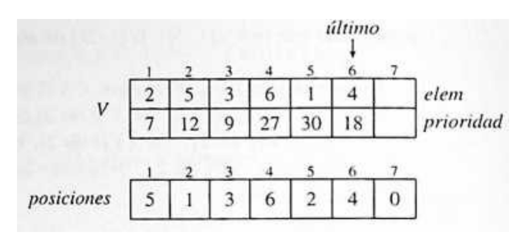
fun montículo-vacío() dev *M : montículo-pares* { 0(A'))
M.último := 0
M .posiciones := [0]
Aun

La función mínimo ahora devuelve un *par,* el que se encuentra en la posición 1 de V.

fun mínimofM : *montículo-pares')* **dev** *p : par* (0(1))
si *M.último* = 0 **entonces** error(Montículo vacío)
si no *p := M.* V [1]
fsi flun

Al eliminar el mínimo hay que marcar que el *elein* en la posición 1 de *V* ya no estará, y que el último se ha movido a esa posición. Ahora la operación hundir-pares recibe el montículo completo, para que modifique tanto V como *posiciones.*

proc eliminar-mín(Af : *montículo-pares)* (0(logN))
si *M.último* = **0 entonces** error(Montículo vacío)
si no
M.posiciones[M.V[l].elem] := 0
*M.*V[1] := *M.V[M.último] ; M ,posiciones[M .V[\].elem] :*= 1



M.último := M.último — 1
hundir-pares(AÍ, 1)
fsi
fproc

El procedimiento auxiliar hundir-pares es prácticamente idéntico, solo que ahora la comparación es entre prioridades, y se utiliza un procedimiento intercambiar que tiene en cuenta las *posiciones.*

proc hundir-pares(M : *montículo-pares,* **e** *j : \..N)* (©(logW))
fin := falso ; *i := j ; k := M.último*
mientras *2* i* < *k* A *-fin* **hacer**
(mínimo de los hijos)
si 2 * *i* + 1 < *k* Ac *M.V[2 * i +* 1],*prioridad < M.V[2 * i],prioridad* **entonces**
m 2 * *i* + 1
si no *tn* := 2 * i
fsi
{ si es necesario, se intercambia con el mínimo de los hijos }
si *M.V[m\.prioridad < M.V[i].prioridad* **entonces**

intercambiarte/, i, m) ; *i := m*
si no *fin* := cierto
fsi
fniientras
fproc
proc intercambiarte/ : *montículo-pares, i, j* : l..Af)
< A/.V[Z], *M.V[j]) := {M.V[j].M.V[i])*
M .posiciones\M .V[i].elem\ := i ; M ,posiciones[M ,V\j].elem\ := j **fproc**

La operación añadir recibe ahora un elemento y una prioridad. El nuevo par se coloca en la posición siguiente a la última, apuntándolo en *posiciones.* Después se utiliza flotar-pares.

proc añadirte/ : *montículo-pares,* **e** *v : \..N.e p : rea!)* (0(logiV))
si *M ,posiciones[v* 0 **entonces** errorfEl elemento ya está)
si no
si *M .último = N* **entonces** error(Montículo lleno)
si no
M último := M.último + 1
A/. *V[M .último],elem :=* v: *M.V[M.último],prioridad := p*
M,posiciones[v] := M.último
flotar-pareste/, *M.último)*
fsi
fsi
fproc
proc flotar-pareste/ : *montículo-pares,* **e** *j* : 1..W) **{ 0(log/V))**
' := J
mientras *i* 1 Ac *M.* V[z *[prioridad < M.V[i* div *2].prioridad* **hacer**

intercambiarte/, *i. i* div 2)
i .= i div 2
fniientras
fproc

La nueva operación modificar recibe el elemento del cual se quiere modificar su prioridad y la nueva prioridad, que puede ser mayor o menor. Para saber qué hay que hacer para colocar correctamente el par modificado, se compara con su padre. Si se ha hecho menor que su padre, entonces se '■flota". En caso contrario, o si no tiene padre, el par se "hunde". Si el elemento a modificar no está, entonces se añade por primera vez.

proc modificarte/ *: montículo-pares, e v : \..N,e p : real)* (O(logA')l
i := M ,posiciones[v]
si *i* = 0 **entonces** añadirte/. u, *p)*
si no
M .V [i],prioridad := p
si *i* 1 *M,V[i],prioridad < M.V[i* div *2].prioridad* **entonces**
flotar-pareste/. *i)*
si no
hundir-pares(A4, *i)*
fsi
fsi
fproc

Este tipo de montículo será útil para resolver los Ejercicios 12.15 y 12.19.

8.9. Implementar un algoritmo que reciba un montículo implementado con la representación del Ejerci**cio** 8.4 y devuelva un árbol binario que represente el mismo montículo. El TAD de los árboles binarios debe tratarse de forma abstracta.

------------------**Solución**-----------------

Para construir el árbol a partir del montículo se utiliza una función recursiva más general que construye un subárbol a partir del montículo y de la posición dentro del vector del elemento que hay que colocar en la raíz.

fun montículo-árbol 1 *(M : montículo)* **dev** *a : árbol-bin*
a := mont-árbol-aux(AÍ, 1) **ffun**

Para construiré! subárbol que tiene como raíz el elemento en la posición *i* de V. primero se comprueba si esa posición está ocupada. Si no lo está, se devuelve el árbol vacío. En otro caso, se construye un árbol que tiene como hijo izquierdo el subárbol construido a partir de la posición 2/. como raíz V|i]. y como hijo derecho el subárbol construido a partir de la posición 2/ + 1 (véase la solución del Ejercicio 8.4 para la explicación detallada de la asignación de posiciones en un vector a los elementos de un montículo).

El coste es lineal con respecto al número de nodos en el árbol construido, por lo que para la primera llamada recursiva desde montículo-árboll el coste es lineal con respecto al número de elementos en el montículo.

fun mont-árbol-aux(A7 : *montículo, i :* 1..W) dev *a : árbol-bin*
si *i* > *M.último* **entonces** *a* := árbol-vacío()
sino *a* := plantar(mont-árbol-aux(AÍ, 2 * í), M.V[í], mont-árbol-aux(A/. 2 * t + I))
fsi
ffun

Este problema también puede resolverse mediante un algoritmo iterativo que realiza un recorrido secuencial por los elementos del montículo según están colocados en el vector, de derecha a izquierda, construyendo subárboles cada vez mayores.

La idea es similar a la de un recorrido en anchura o por niveles (Ejercicio 6.12). solo que en este caso el proceso se hará "al revés": se irá de las hojas hacia la raíz, cada nivel se recorrerá de derecha a izquierda, y del recorrido obtendremos el árbol. En un recorrido de esta forma, la primera mitad de los elementos son hojas. Estas hojas se colocarán en una cola (de árboles), para que estén disponibles cuando se necesiten. Los restantes elementos serán nodos internos, cuyos hijos serán los dos siguientes árboles de la cola. El proceso se repite hasta que se hayan recorrido todos los elementos del montículo.

Hay que tener cuidado, porque al ser el montículo un árbol *semicompleto.* puede que el primer nodo interno en el recorrido tenga solo un hijo, el hijo izquierdo. Esto

se resuelve añadiendo inicialmente un árbol vacío a la cola si es necesario. La complejidad del algoritmo es lineal con respecto al número de elementos en el montículo, o sea, el número de nodos del árbol resultado.

fun montículo-árbol2(*M : montículo)*
dev *a : árbol-bin*
var *C : cola[árbol-bin], iz. dr : árbol-bin*
C := cola-vacía()
m := M.último div 2
si par (Ai *.último)* **entonces** añadirte, árbol-vacío()) **fsi**
para *i = M.último* **hasta** *m* + I **paso** - 1 **hacer**
añadir(C, plantar(árbol-vacío(). *M.V*[/], árbol-vacío()))
fpara
para *i = ni* **hasta 1 paso - I hacer** *dr* *:=* **primero(C) ; avanzar(C) íz := primero(C) : avanzar(C) añadir(C, plantarfíz.** *M ,V[i].dr))*
fpara
a := pñmero(C) ; avanzar(C)
ffun

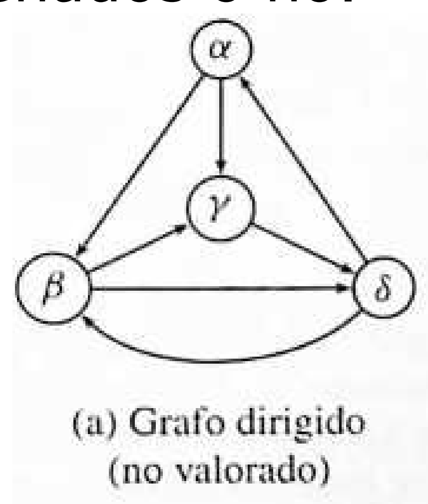
Capitulo 9

9. GRAFOS

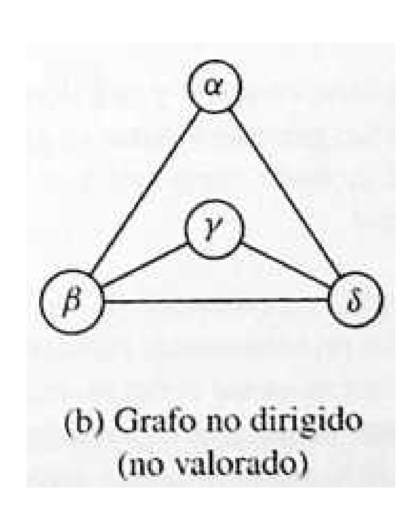
INTRO

En este capítulo vamos a estudiar las estructuras relaciónales que generalizan las estructuras arbóreas de los temas anteriores. Un **GRAFO** *G* se define como un par de conjuntos finitos *(V. A).* donde V es el conjunto de *vértices y A* C *V* x V es un conjunto de pares de vértices llamados *aristas.* La existencia de aristas entre vértices da lugar a una relación binaria sobre el dominio de los vértices. Se distinguen diferentes tipos de grafos:

- **dirigidos,** cuando los pares de las aristas están ordenados, es decir, la relación dada por el grafo no es simétrica;
- **no dirigidos,** cuando los pares de las aristas no están ordenados, es decir, la relación es simétrica;
- **valorados,** cuando las aristas tienen asociado algún valor (peso, longitud, etc.) y sus vértices pueden estar ordenados o no.



(a) Grafo dirigido (no valorado)



(b) Grafo no dirigido (no valorado)

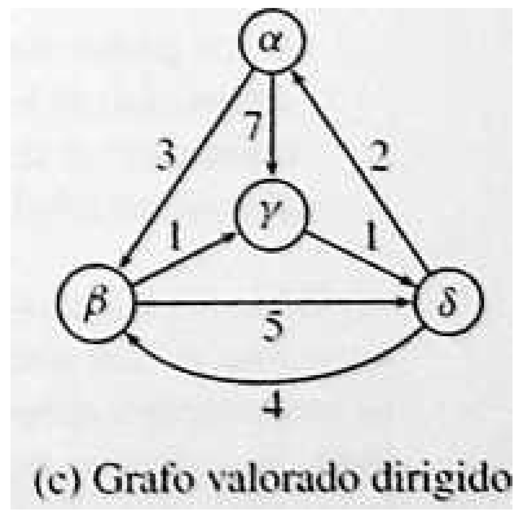


Figura 9.1; Ejemplos de grafos.

La Figura 9.1 contiene un ejemplo de cada uno de estos tipos de grafos. donde un elemento (r. r’) £ *A* se representa mediante una Hecha del vértice v al v' en el caso de grafos dirigidos y mediante una línea entre ambos si el grafo es no dirigido. En el grafo (c) las aristas están valoradas con números naturales que aparecen junto a la flecha correspondiente.

En los grafos dirigidos, para una arista (*i>. w). v es* el *origen, ir* es el *destino* y se dice que *ir* es *adyacente* a v. El *grado de entrada* de un vértice v es el número de aristas en el grafo de las cuales v es el destino, mientras que el *grado de salida* corresponde al número de aristas de las cuales r es el origen. En el caso de los gratos no dirigidos, se habla simplemente de *grado* de un vértice y *extremos* de una arista, siendo ambos adyacentes entre sí.

A continuación definimos algunos términos relativos a los grafos y que aparecerán en los ejercicios de este capítulo.

Subgrafb: Se obtiene eliminando algunos vértices y aristas de un grafo; así *C* = (V". *A'}* es *subgrafo de G = {V.*

A), denotado por G' C G, si V' C V, A' C A y G' está bien definido como grafo.

Camino: Es una secuencia de vértices adyacentes, es decir, para un grafo G = (V. A) la secuencia (vi, *V2* *v„*) con *n* > 1 y *v, e* V (para *i* entre 1 y n) es un *camino* en G si Vi I < *i* < *n :* *{»i,* ti,+i) e A.

La *longitud* de un camino es el número de aristas que lo forman, es decir, la longitud de un camino
< ti i. t>2 ti„) es n — I. Por ejemplo, en el grafo de la Figura 9.1 (a) la secuencia *(a. y. S. 0} es* un camino de longitud 3.

La *distancia mínima* entre dos vértices de un grafo. A(i>, ut). se define como la longitud del camino más corto que parte de v y llega hasta w. Por ejemplo, en el grafo de la Figura 9.1 (a) se tiene A(a.ó) = 2.

En muchas ocasiones, en los grafos valorados los valores asociados a las aristas son números no negativos que se interpretan como costes; en esos casos, se dice que el *coste* de un camino es la suma total de los costes asociados a las aristas que lo forman. El *coste mínimo* entre dos vértices. F(v. ui). se define como el coste del camino de u a tu cuyas aristas sumen el menor valor Por ejemplo, en el grafo de la Figura 9.1 (c) la secuencia (a. *y. 8. p*) es un camino de coste 12. y Fio. ó) = 5. que corresponde al coste del camino *(a. p, y. 8).*

Ciclo: Es un camino de longitud no nula que comienza y termina en el mismo vértice. Se dice que un grafo es *acíclico* cuando no tiene ciclos. El grafo de la Figura 9.1 (a) no es acíclico porque, por ejemplo, la secuencia *(a. y, 8. a*) forma un ciclo.

Los ciclos que pasan exactamente una vez por cada vértice se denominan ***hamiltonianos***. Por ejemplo. en el grafo de la Figura 9.1 (a) la secuencia (o. *p, y. 8, a)* es un ciclo hamiltoniano.

Aunque en los ejemplos de grafos que se darán a lo largo de este capítulo no aparecen *auto-aristas,* es decir, aristas cuyo origen y destino coinciden, sí son posibles según la definición que hemos dado de grafos. Estas auto-aristas no suelen representar mayor problema que los ciclos y. salvo que se indique lo contrario o se requiera que los grafos sean acíclicos, los algoritmos que se dan en las soluciones a los ejercicios sobre grafos son correctos aunque el grafo contenga este tipo de aristas.

Conexión: Un grafo no dirigido es *conexo* si para cada par de vértices diferentes t> y *w* existe un camino desde *v* a tu; un grafo dirigido es ***fuertemente conexo*** si para cada par de vértices diferentes u y *w* existe un camino desde *v* a *w* y un camino desde *w* a *v.* Los ejemplos de la Figura 9.1 (a) y (c) son grafos fuertemente conexos; el de la Figura 9.1(b) es conexo.

Si un grafo dirigido no es fuertemente

conexo, los mayores conjuntos de vértices tales que los subgrafos inducidos son fuertemente conexos se denominan *componentes fuertemente conexas* del grafo.

A los grafos no dirigidos, conexos y acíclicos se los denomina *árboles libres.* Estos árboles se diferencian de los árboles generales vistos en el Capítulo 6 en que no tienen raíz y los hijos no están ordenados; si se elige un nodo como raíz y se ordenan de alguna forma los hijos de cada nodo, se obtiene un árbol general.

Ofrecemos a continuación una colección de ejercicios sobre la especificación y la representación de los grafos. así como sobre los problemas más básicos sobre grafos: recorridos sobre grafos. determinación de componentes conexas y fuertemente conexas, etc. Sin embargo, las soluciones que ofrecemos en este libro para algunos problemas sobre grafos corresponden a técnicas que se detallan en los capítulos de la segunda parte del libro. Pensamos que estas soluciones se podrán comprender más fácilmente tras la introducción a la correspondiente técnica y en relación con otros ejercicios de naturaleza similar; por ello hemos optado por no incluirlos en este capítulo dedicado a los grafos. En concreto, problemas tales como obtener un árbol de recubrimiento de coste mínimo (algoritmos de Pnm y de Kruskal) o determinar los caminos de coste

mínimo desde un vértice a todos los demás (algoritmo de Dijkstra). se resuelven mediante algoritmos *voraces,* que se detallan en el Capítulo 12 (Ejercicios 12.13 a 12.21). En cambio, determinar los caminos de coste mínimo entre cada par de vértices (algoritmo de Floyd) o calcular la clausura reflexiva y transitiva de la relación dada por un grato (algoritmo de Warshall) (Ejercicio 13.9) son ejemplos de aplicación de la técnica de *programación dinámica* (Capítulo 13). Por último, problemas como la obtención de ciclos hamiltonianos de coste mínimo (Ejercicios 14.14 y 15.8), el coloreado de grafos (Ejercicio 14.7). etc . requieren la exploración exhaustiva de espacios de soluciones, que nosotros realizamos mediante las técnicas de *vuelta atrás* (Capítulo 14) y *ramificación y poda* (Capítulo 15).

EJERCICIOS

9.1. Especificar un TAD para describir los *grafos* con vértices pertenecientes a un tipo dado como parámetro, con las siguientes operaciones:

- crear el grafo vacío,
- añadir un vértice,
- añadir una arista,
- eliminar un vértice y todas las aristas de las que es origen o destino.

o eliminar una arista.

o determinar si un vértice pertenece a un grafo,

o determinar si una arista pertenece a un grafo.

o determinar si un grafo es vacío, y
o obtener el conjunto de vértices adyacentes a uno dado.

(a) Especificar los *grafos dirigidos.*
(b) Especificar los *grafos no dirigidos.*

Solución:

El tipo parámetro para los vértices tan solo requiere que tenga definida una relación de igualdad entre sus elementos,

parámetro *VÉRTICES*
usa *BOOLEANOS*
tipos *vértice*
operaciones
_ == _ *vértice vértice —» bool*
: vértice vértice —► bool
variables
V. w : vértice
ecuaciones
(t> == ut) = cierto <= *v* = ir
v = *ut* <= (o == te) = cierto
vfiw = -•(u == *w)*
fparámetro

Nótese que el parámetro *VERTICES* es esencialmente el mismo que *ELEM=* definido en la Sección 1.1.5, aunque ahora el tipo definido es *vértice* en lugar de *elemento.*

La elección de constructoras es la misma para grafos dirigidos y no dirigidos: se parte de un grato vacío (grafo-vacío) y se añaden los vértices (añ-vértice) y aristas (añ-arista) deseados. Obviamente, no se puede añadir una arista sin haber añadido previamente los vértices origen

y destino: por lo demás, el orden en que se añaden los vértices y las aristas no es relevante, y añadir varias veces el mismo elemento (vértice o arista) no tiene ningún efecto. Por tanto, las constructoras *no son libres* y se darán las correspondientes ecuaciones de equivalencia.

La operación observadora que devuelve los vértices adyacentes a uno dado solo tiene sentido si el vértice está en el grafo. por lo que se define como parcial. Las restantes operaciones modificadoras y observadoras son totales, porque eliminar un elemento (vértice o arista) que no esté en el grafo no tiene efecto sobre el grafo.

A continuación damos la signatura de las operaciones, que es común para los grafos dirigidos y no dirigidos (si bien habría que dar un nombre diferente a cada especificación).

especificación *GRAFOS[VÉRTICES]*
usa *BOOLEANOS. CONJUNTOS[VÉRTICES]*
tipos *grafo*
operaciones
grato-vacío : —> *grafo* (constructora)
añ-vértice *: vértice grafo —♦ grafo* (constructora i
añ-arista *vértice vértice grafo —grafo* (constructora)
elim-vértice *: vértice grafo —> grafo*
elim-arista *: vértice vértice grafo —>*

grafo
está-vértice? *: vértice grafo —> bool*
está-arista? *: vértice vértice grafo —> bool*
es-grafo-vacío? : *grafo —♦ bool*
adyacentes *: vértice grafo— conjunto[vértice]*

variables

v, w, v', w' : vértice
g : grafo

La diferencia entre grafos dirigidos y no dirigidos se refleja en las ecuaciones de equivalencia entre términos construidos y en el comportamiento de algunas operaciones, por lo que daremos las ecuaciones correspondientes por separado. En el Apartado (b) no se repiten las especificaciones de las operaciones que no cambian con respecto al Apartado (a).

Apartado (a)

Vemos en primer lugar las ecuaciones para grafos dirigidos. Como se ha comentado anteriormente, la operación añ-arista da error si los vértices no están previamente incluidos en el grafo. pero los vértices y aristas se pueden añadir en cualquier orden que respete dicha restricción.

ecuaciones

añ-arista(v. u>. g) = error <= ->está-vértice?(v, g) v -▪está-vértice?(*w.* g)
añ-vértice(u, añ-vértice(u. g)) = añ-vértice(ti, g)

añ-vértice(iz), añ-vértice(v, g)) = añ-vértice(u, añ-vértice(u). g))

añ-arista(i). *w.* añ-arista(u. *w. g))* = añ-arista(u, w. g)

añ-arista(v. *w, añ-arista(v', w'.* g)) = añ-arista(u', *w',* añ-arista(i>, *tu. g))*

añ-arista(v, *w.* añ-vértice(u'. g)) = añ-vértice(i)', añ-arista(u, *w,* g))

<= *(v'* u A *v' w) v* (está-vértice?(v. g) A está-vértice?(tu, g))

Como hacemos siempre que tenemos constructoras parciales, a partir de ahora consideraremos que los términos formados por constructoras que aparecen en las ecuaciones son correctos.

Para eliminar un vértice hay que eliminar todas sus apariciones en la construcción del grafo. tanto si corresponde a añadir dicho vértice como a una arista que lo tenga como origen o como destino.

elim-vértice(*v.* grafo-vacío) = grafo-vacío
elim-vértice(u, añ-vértice(u, g)) = elim-vértice(u, *g)*
elim-vértice(v, añ-vértice(u), g)) = añ-vértice(ui, elim-vértice(r. g)) <= r == *w*
elim-vértice(i). añ-arista(u, *w,* g)) = elim-vértice(i'. *g)*
elim-vértice(u, añ-ansta(w, *v,* g))= elim-vértice(v. g)
elim-vértice(i), añ-arista(i/, *w′,* g)) = añ-arista(v'. u/, elim-vértice(r. g)) <= *v v' A v w'*

Una arista hay que eliminarla tantas veces como se ha añadido.

elim-arista(u, *w.* grafo-vacío) = grafo-vacío
elim-aristafi;. *w.* añ-vértice(v', g)) = añ-vértice(v', elim-aristaív. *w.* g))
elim-arista(v, *w.* añ-arista(u, *w,* g)) = elim-arista(v, *w.* g)
elim-aristafv, *w.* añ-arista(v', *w'.* g)) = añ-arista(v'. *w'.* elim-aristafi’, *w.* g))
<= v ¿ t>' v w / u'

El comportamiento de los tres predicados viene dado mediante la distinción de casos de las tres constructoras.

está-vértice[9](u. grafo-vacío) = falso
está-vértice?(r>. añ-vértice(u), g)) = v == *w v* está-vértice?(u. g)
está-vértice?(u. añ-arista(i)'. *w',* g)) = está-vértice?(u, g)
está-arista?(v. u>, grafo-vacío) =

falso

está-arista?(u. *w,* añ-vérticefi/, g)) =

está-arista?(v, *w.* g)

está-arista?(v, *w.* añ-arista(i)'. *w'.* g))

=*(v == v'* A *w ==w')* v

está-arista?(r.ir. g)

es-grafo-vacío? (grafo-vacío) = cierto

es-grafo-vacío?(añ-vértice(u, g)) =

falso

es-grafo-vacío?(añ-arista(v, *w.* g)) =

falso

Los vértices adyacentes a *v* son todos aquellos que son destino de una arista cuyo origen es v. Si el vértice no pertenece al grafo, se obtiene un error. Esto incluye al grafo vacío como caso particular, por lo que no explicitaremos una ecuación para la constructora grafo-vacío, pero se distinguen los casos de las otras dos constructoras: se pueden ignorar otros vértices añadidos al grafo que no sean el vértice para el que se calculan los adyacentes (llamémoslo u); también en el caso que el vértice añadido sea justamente *v, pero* este ya estaba previamente en el grafo. En cambio, cuando se detecta la primera inclusión de r en el grafo, ya se sabe que no habrá más vértices adyacentes a v (se tiene el conjunto vacío). Cada vez que se encuentra una arista con origen *v. se* añade el vértice destino al conjunto de adyacentes. Nótese que las repeticiones no afectan al resultado puesto que la operación añadir de conjuntos también es idempolente.

adyacentes(u, g) = error <= -■está-vértice?(v, g)
adyacentes(u. añ-vértice(u, g)) = cjto-vacío <= -■está-vértice? (u, g)
adyacentes(i), añ-vértice(u). g)) = adyacentes (v, g) <= *v* 7= u> V está-vértice?(r. g)
adyacentes(u, añ-arista(u. *ir, g)) =* añadir(u>. adyacentes(r, g))
adyacentes(i>. añ-arista(i>', *w',* g)) = adyacentes(u. g) s= 1’7=1”
fespeciíicación

Apartado (b)----------------

En un grafo no dirigido, el origen y el destino de una arista son intercambiables; así que añadimos la siguiente ecuación al conjunto de ecuaciones de equivalencia dado para los gratos dirigidos.

añ-arista(v, *w. g)* = añ-arista(u), it, g)

El comportamiento de la operación elim-vértice es el mismo que en los grafos dirigidos, puesto que se eliminan tanto las aristas de entrada al vértice, como las de salida. Pero al eliminar una arista habrá que considerar ahora la posibilidad de que los extremos estén en el orden contrario.

elim-aristafv. i», grafo-vacio) = grafo-vacío
elim-aristafv, *w.* añ-vértice(v'. g)) =
añ-vértice(i)',elim-aristafv. *w. g))*
elim-aristafv, *w.* añ-arista(v, *tu.* g)) =
elim-aristafv, *tu, g)*
elim-aristafv. tv. añ-aristafur. v. g)) =

elim-aristafv. *w. g)*

elim-aristafv. tv. añ-aristafv'. *w'.* g)) = añ-aristafv', *w',* elim-aristafu, *u>.* g))

<= (v *v'* *v* *w* tv') A (v *w'* v *w* / *v')*

En los predicados solo cambia el comportamiento de está-arista?; como en la operación anterior, hay que considerar la posibilidad de que los extremos aparezcan en el orden contrario.

está-arista?(v, iv. grafo-vacío) = falso

está-arista?(v. tv, añ-vérticefv'. g)) = está-arista?(v, u>, g)

está-arista?(v. *w.* añ-arista(u', *w'.* g)) = (v == *v'* *A* *w* == *w') V (v* == *w'* *A* *w* == *v')*

v está-arista? (i>. *w,* g)

También cambia el comportamiento de adyacentes, porque ahora los vértices adyacentes a v son todos aquellos que son extremo de una arista cuyo otro extremo es v. Cada vez que se encuentra una arista con *v* en un extremo, se añade el otro vértice al conjunto de adyacentes.

adyacentes(*v.* g) = error <= --está-vértice?(v. g)

adyacentesfv, añ-vértice(r. g)) = cjto-vacío <= -■está-vértice?(v, g)

adyacentesfv. añ-vérticefiv, g)) = adyacentesfv. g) <= *vw* v está-vértice?(v, g)

adyacentesfv. añ-aristafv. un g)) = añadirftv. adyacentesfv. g))

adyacentesfv. añ-aristaftv. v. g)) = añadirftv. adyacentesfv. g))

adyacentesfv. añ-arista(v', *vi'. g))* =

adyacentesfv. g) <= vv'A v *w'*

Extender el TAD de los grafos con una operación que dados dos vértices determine si el segundo es accesible desde el primero.

(a) Especificarla para grafos dirigidos.

(b) Especificarla para grafos no dirigidos.

------------------Solución-----------------

Consideramos la nueva operación como *parcial,* porque no tiene sentido si los vértices no pertenecen al grafo.

La siguiente signatura es válida tanto para grafos dirigidos como no dirigidos.

especificación *GRAFOS+[VÉRTICES]*
usa *GRAFOSfVÉRTICES], BOOLEANOS*
operaciones
accesible? : *vértice vértice grafo —>* $_p$ *bool* **variables**
v.w.v'.w't vértice
g : *grafo*

La especificación del comportamiento de esta nueva operación se hace por distinción de casos sobre las tres constructoras de los grafos (grafo-vacío. añ-vértice y añ-arista) y es ligeramente diferente para cada tipo de grafos.

Apartado (a)

Comenzamos con los grafos dirigidos Si alguno de los vértices no pertenece al grato, se obtiene un error. Esto incluye al grafo vacío como caso particular, por lo que no explicitaremos una ecuación para la constructora grato-vacío.

Se pueden ignorar los vértices añadidos a un grafo mientras los vértices objeto del análisis de accesibilidad (llamémoslos *v y w)* sigan estando en el grafo; pero si al quitar un vértice añadido *v′* resulta que o bien *v o* bien *w* ya no está en el grafo. eso significa que la accesibilidad de ir desde r solo es posible si ambos vértices son el mismo y además coinciden con *v′,* porque en el grafo restante (que se supone bien construido) no puede haber aristas que entren o salgan del vértice "desaparecido". Si encontramos una arista con 1/ como origen y *w′* como destino, o bien *w* es accesible desde r por un camino que no use esa arista y por tanto es accesible en el resto del grafo, o bien ir es accesible desde r por un camino que use esa arista y en ese caso *v′* tiene que ser accesible desde i> y *w* tiene que ser accesible desde *w′* en el resto del grafo.

ecuaciones

accesible?(u. *w. g)* — error <= -■está-vértice?(i'. g) v -■está-vértice?(u’. g)

accesible?(i>. *w.* añ-vértice(u', g))) = accesible?(v. *w.* g)

<= está-vértice?(u. g) A

está-vértice?(ic. g)

accesible?(u. *w.* añ-vértice(i/. g))) = *v* == *v' /\ w* == *v'*
<= -■(está-vértice?(v. g) A
está-vértice?! ir. gl)
accesible?! i[1], *w.* añ-arista(i>', *w'. g)) = accesible?(u. un g) v*
(accesible?(v, v'. g) A
accesible?! ir', ir. g)) fespecificación

La relación de accesibilidad coincide con la clausura reflexiva y transitiva de la relación dada por un grafo. En el Apartado (c) del Ejercicio 13.9 se verá el algoritmo de Warshall que calcula esta clausura para un grafo dado

Apartado (b)

En los grafos no dirigidos cada arista puede considerarse en los dos sentidos, así que basta modificar la última ecuación del apartado anterior para considerar la arista en ambos sentidos.

ecuaciones
accesible?) r>. *w. g)* = error <= -■está-vértice?(r. g) v -■está-vértice?(ir. g)
accesible?(u, *w.* añ-vertice(u', g))) =
accesible?(v. u’. g)
<= está-vértice?(i>. g) A
está-vértice?(tr. g)
accesible?(u. *w.* añ-vértice(i/. g))) = t> == *v'* A *w* == r
<= -i(está-vértice?(u. g) A
está-vértice?(ir. gl) accesible?! *v.* u>, añ-arista(u'. ir'. g)) = accesible?(i>, ir.g) v
(accesible?(i>, r'. g) A
accesible?) ir', te. gl) v

(accesible?(f. ti ', g) A
accesible[7])r', ir. gl) fespecificación

Diseñar una representación del TAD de los grafos utilizando una *matriz de adyacencia* que para cada par de vértices indique si son adyacentes, es decir, si existe la arista correspondiente. Implementar las operaciones especificadas en el Ejercicio 9.1.

(a) Representar los grafos dirigidos.

(b) Representar los grafos no dirigidos.

------------------Solución------------------

Apartado (a)

La idea es representar la matriz de adyacencia mediante un vector tridimensional de booleanos, cuyos índices serán números naturales positivos, porque numeraremos los vértices según el orden de adición al grafo. Los elementos correspondientes a los vértices se almacenarán en un vector *vértice.* De esta forma, la fila i-ésima de la matriz corresponde a las aristas cuyo origen es el vértice i-ésimo, mientras que la columna í-ésima corresponde a las aristas cuyo destino es el vértice í-ésimo.

Por ejemplo, al grafo de la Figura 9.1 (a) (suponiendo que los vértices se han añadido al grafo en el orden alfabético *a. fi. y. S)* le corresponde la matriz de adyacencia siguiente:

` 0 1 1 0 `
0 0 1 1
0 0 0 1
1 1 0 0 _

donde hemos representado los valores booleanos con 0 (falso) y 1 (cierto).

Como ocurre con la mayoría de las

representaciones estáticas de estructuras cuyo tamaño es variable, es necesario definir a priori un tamaño máximo (en este caso, la cantidad de vértices) y llevar un indicador del tamaño actual (véase la Sección 2.1). Por tanto, el tipo representante es el siguiente:

tipos
grafo **= reg**
matriz[l..N. 1..N] de *bool*
vértice[i..N] de *vértice*
tamaño : 0..N
freg
ftipos

Cuando el grafo esté vacío, *tamaño* tendrá el valor cero.

fun grafo-vacío() **dev** *g : grafo* { 0(1))
g.tamaño := 0
ffun

Para añadir un vértice, una vez se ha comprobado que el vértice todavía no pertenece al grafo y que hay hueco para un nuevo vértice, el tamaño se incrementa en uno y la fila y columna correspondientes se inicializan a falso.

proc añ-vértice(e u : *vértice, g : grafo)* { 0(g.*tamaño) }*
(b, p) := buscar(g.*vértice, g.tamaño. v)*
si *-•b* **entonces**
si *g.tamaño* = *N* **entonces** error(Espacio insuficiente)
si no
g.tamaño := g.tamaño + 1
g.vértice[g.tamaño] := v
g.matriz[g.tamaño, i..g.tamaño] := [falso]

g.matriz[i..g.tamaño, g.tamaño]
[falso]
fsi
fsi
fproc

Para comprobar la existencia del vértice hemos utilizado la función buscar, que localiza un vértice en el vector *vértice* y devuelve el índice que le corresponde (porque será útil más adelante), o bien indica que la búsqueda no ha tenido éxito.

fun buscarf V[l . /V | **de** *vértice, t :O..N,v : vértice)* **dev (***h : bool. p* **: 1...V** | *0t g.tamaño))*
b := falso : *i :=* 1
mientras -•*b r\ i* < *t* **hacer**
si V[i] = *v* **entonces** *b :=* cierto ; *p := i*
si no *i :=* *i* + 1
fsi
fmientras ffun

Al añadir una arista hay que comprobar primero que el origen y el destino pertenecen al grato. Utilizamos la función buscar que acabamos de detallar, que además nos proporciona los índices correspondientes.

proc añ-ansta(e v, *tu : vértice, g : grato) { 0(g.tamaño))*
(b\.p\ ) := buscar(g.vértice, g.tamaño. v)
(t>2. P2) buscarlg.vértice, g.tamaño. w)
si -■/?) v ^*l>2* **entonces** error(Vértices desconocidos)
sino *g.malrizlpi. P2]* .= cierto

fsi
fproc

Antes de eliminar un vértice hay que comprobar que está en el grafo y averiguar su índice. Eliminar un vértice significa eliminar también todas las aristas que entren o salgan de él. pero no pondremos a falso los valores en la matriz de adyacencia, sino que eliminaremos por completo la correspondiente tila y columna, reduciendo el tamaño de la parte ocupada de la matriz y desplazando hacia índices anteriores la información. Similar desplazamiento debe sufrir el vector *vértice,* para cubrir el hueco dejado por el vértice eliminado.

proc elim-vértice(e *v : vértice, g : grato)*
{ 0 (g.tamaño²))
{b. p) := buscar(g.vértice. g.tamaño, v)
si *b* **entonces**
para í = p + 1 **hasta** *g.tamaño* **hacer**
g.vértice[i — 1] := g.vértícefí]
g.matrizli — 1. 1 *..g.tamaño] :=*
g.matriz[i. 1..g.tamaño]
fpara
para / = p + 1 **hasta** *g.tamaño* **hacer**
g.matriz[\..g.tamaño - l,t - 1] :=
g.matriz[\..g.tamaño - l.í]
fpara
g.tamaño := g.tamaño — 1
fsi
fproc

Eliminar una arista requiere averiguar los índices correspondientes a los extremos y poner a falso la entrada en la

matriz de adyacencia. Si alguno de los extremos no está en el grafo, no se hace nada.

proc elim-arista(e *v. iv : vértice, g : grato)* j *0(g.tamaño) }*
(b\.pi) := buscarfg.*vértice, g.tamaño, v)*
(bz. P2) '■= buscar(g.vértice, g.tamaño. w)
si *b\ r\ b2* **entonces** *g.matriz[pi. P2]* '■= **falso fsi fproc**

Para determinar si un vértice o una arista están en el grafo. recurrimos otra vez a la función buscar.

fun está-vértice? (u . *vértice, g : grato)* **dev** *b : bool { 0 (g.tamaño) }*
(b, p) := buscar(g.vértice, g.tamaño, v)
ffun
fun está-arista?(i', *w : vértice, g : grato)* **dev** *b : bool { &(g.tamaño) }*
{b\. p\) := buscar(g. *vértice, g.tamaño,* t>)
(¿2- *Pi*) := buscartg.vérríce. *g.tamaño. w)*
si -•Z»i v -¿n **entonces** *b* := falso
sino *b := g.matriz[p\. P2]*
fsi
ffun

El grafo es vacío si el tamaño es 0.

fun es-grafo-vacío?(g : *grato)* **dev** *b : bool* (0(1))
b := (g.tamaño = 0) **ffun**

Para obtener el conjunto de vértices adyacentes a uno dado, debemos primero localizar el vértice para después recorrer la fila correspondiente. Si el vértice no

estuviera en el grafo. se produciría un error.

En la mayoría de los algoritmos que trabajan con gratos, lo que se necesita es poder recorrer todos los vértices adyacentes a un vértice dado, pero los conjuntos no lo permiten. Por eso vamos a implementar la operación adyacentes de forma que devuelva una *lista* de vértices en lugar de un conjunto.

fun adyacentes(v : *vértice, g : grato)*
dev *l : lista[vérticei (&(g.tamaño))*
/ := lista-vacía()
(b. p) := buscarfg.*vértice, g.tamaño.* u)
si *b* **entonces**
para *i* = 1 **hasta** *g.tamaño* **hacer**
si *g.matrizlp.* í] **entonces** añadir-der(/. g.vértíce[íj) **fsi**
fpara
si no error(Vértice desconocido)
fsi
ffun

Todas estas operaciones (salvo grato-vacío y es-grafo-vacío?) realizan llamadas a la función buscar, cuyo coste es lineal con respecto al número de vértices en el grafo. De esta forma el coste de las operaciones añ-vértice. añ-arista, elim-arista. está-vértice?, está-arista? y adyacentes resulta ser también lineal con respecto al número de vértices, y el de la operación elim-vértice es cuadrático. pues requiere copiar las filas y columnas posteriores al vértice eliminado. Estos costes pueden parecer excesivamente elevados, pero es que la implementación

que acabamos de detallar considera grafos totalmente *dinámicos* en los que se pueden añadir y eliminar tanto vértices como aristas. Sin embargo, en los ejercicios que veremos a continuación, el conjunto de vértices estará fijado desde el principio y no variará (utilizaremos la notación *grafo[n]* para indicar que se trata de un grafo con *n* vértices). Además, la naturaleza de los vértices será irrelevante para los problemas, por lo que serán simplemente representados por su número de orden. De esta forma, la representación dada se reduce a la matriz (de adyacencia) y se eliminan el *tamaño* y el vector *vértice.* Bajo estos supuestos, las operaciones grafo-vacio, añ-vértice, elim-vértice, está-vértice? y es-grafo-vacío? no tendrán razón de ser, y el resto de las operaciones se implementan mediante accesos a la matriz de adyacencia, resultando añ-arista. elim-arista y está-arista? de coste constante y solo adyacentes de coste lineal con respecto al número de vértices (véase la implementación de grafos valorados del Ejercicio 9.6).

En cuanto al coste en espacio de esta representación, está en $0(A^2)$. o en $0(n^2)$ si fijamos en *n* el número de vértices del grafo. En cualquier caso, el coste es independiente del número de aristas, el cual oscila entre 0 (no hay ninguna arista) y *ir* (hay una arista entre cada par de vértices y en cada sentido). En el

Ejercicio 9.4 se ofrece una representación alternativa de los grafos cuyo coste en espacio sí depende del número de aristas. Completamos esta representación de los grafos con una operación para *copiar* grafos (véase la discusión en la Sección 2.1.3). El algoritmo se reduce a realizar una asignación de la parte ocupada de los

campos del registro que representa el grafo; el coste de la copia resulta cuadrático respecto al **número de** vértices.

fun copiar-grafo(g : *grafo)* **dcv** *f : grafo (©(g.tamaño²) |*
f.tamaño := g.tamaño
f.vértice]\..f.tamaño] ■.= g.vértice] I *..g.tamaño]*
f.matriz]).. f.tamaño, \..f.tamaño] := g.matriz]) ■-g.tamaño. I..g.tamaño] **ffun**

Apartado (b)----------------

En el caso de los grafos no dirigidos el tipo representante no cambia, aunque ahora la matriz de adyacencia es *simétrica,* ya que añadir una arista (v. *iv}* equivale a añadir también la arista (ir. r).

Por ejemplo, al grafo de la Figura 9. l(b) le corresponde la matriz de adyacencia siguiente (suponiendo que los vértices se han añadido al grafo en el orden alfabético *a. 0. y. S):*

' 0 1 0 1
10 11
0 10 1
1110

La implementación de añ-arista queda como sigue:

proc añ-arista(e *v, w ; vértice, g : grafo) (©(g.tamaño))*
(/>i. pi) := *buscarfg.vértice, g.tamaño, v)*
(bz. 1’2) ⁝= buscar(g.*vértice, g.tamaño, w)*

si -■/>| v **entonces** error(Vértices desconocidos)
sino *g.matriz]pi* • P?J := cierto; *g.matrízlpz- P* I := cierto
fsi
fproc

Al eliminar una arista habrá que poner a falso dos entradas en la matriz de adyacencia.

proc elim-arista(e u, *w : vértice, g : grafo) (©(g.tamaño))*
(¿l.pi) := buscarfg.*vértice, g.tamaño, v)*
(bi, pz) '■= buscar(g *.vértice, g.tamaño,* ir)
si /?] A *bi* **entonces** *g.matriz]pt*. pj] := falso; *g.matriz]p2, pi* 1 := falso **fsi fproc**

El resto de las operaciones se implementan igual que para los grafos dirigidos, y las consideraciones sobre los costes son igualmente aplicables, aunque en este caso cabe la posibilidad de reducir el coste en espacio, almacenando solamente la mitad de la matriz (el triángulo superior o el inferior con respecto a la diagonal principal) aunque ello no varía el orden de complejidad de dicho coste.

Diseñar una representación del TAD de los grafos utilizando *fistos de adyacentes.* donde para cada vértice en el grafo se tiene una lista con todos sus vértices adyacentes. Implementar las operaciones especificadas en el Ejercicio 9.1.

(a) Representar los grafos dirigidos.

(b) Representar los gratos no dirigidos.

------------------Solución---------------

Apartado (a)

Siguiendo algunas de ias ideas expuestas en la solución del Ejercicio 9.3. numeraremos los vértices según el orden de adición al grafo y los elementos correspondientes a los vértices se almacenarán en un vector *vértice.* De esta forma, las listas de adyacentes se pueden organizar en un vector de listas cuyos índices serán números naturales positivos y cuyo tamaño limitará el número máximo de vértices en el grafo; llevaremos un indicador del tamaño actual. Por tanto, el tipo representante es el siguiente:

tipos
grafo = **reg**
lista-ad[\..N] **de** *lista[vértice]*
vértice[\..N] **de** *vértice*
tamaño: 0..N
freg
ftipos

Cuando el grafo es vacío, *tamaño* tendrá el valor cero.

fun grafo-vacío() **dev** *g : grato* { 0(1))
g.tamaño := 0
ffun

Para añadir un vértice, una vez se ha comprobado que el vértice no está ya incluido en el grafo y que hay hueco para un nuevo vértice, el tamaño se incrementa en uno; la lista correspondiente estará vacía.

proc añ-vértice(c *v : vértice, g : grafo)* *(Q(g.tamaño))*

(b.p) buscarfg.vértice, g.tamaño, v)

si -•ó entonces

si *g.tamaño = N* **entonces**

error(Espacio insuficiente)

si no

g.tamaño := g.tamaño + 1

g.vértice[g.tamaño] := v

g.lista-ad[g.tamaño] := lista-vacía()

fsi

fsi

fproc

Para comprobar los vértices hemos utilizado la función buscar, definida en la solución del Ejercicio 9.3.

Al añadir una arista hay que comprobar primero que los dos extremos existen en el grafo. Una vez realizada la comprobación, el vértice destino de la arista se añade (por ejemplo, por la izquierda) a la lista de adyacentes del vértice origen (pero solamente en el caso de que el destino no esté previamente incluido).

proc añ-arista(e *v, w ; vértice, g ; grafo)* *(Q(g.tamaño))*

(í>l,pi) := buscar(g.*vértice, g.tamaño, v)*

{bl.Pí) '■= *buscarfg.vértice. g.tamaño, w)*

si -■¿i v *-'bz* **entonces** error(Vértices desconocidos)

sino si ~’está?(ui, *g.lista-ad[pi])* **entonces** añadir-izq(u), *g.lista-ad[p\])*

```
fsi
     fsi
   fproc
```

La operación está? para buscar un vértice en una lista está implementada en el Ejercicio 5.5.

Antes de eliminar un vértice hay que comprobar que está en el grafo y averiguar su índice. Eliminar un vértice significa eliminar también todas las aristas que entren o salgan de él; en el caso de las aristas salientes basta eliminar la lista de adyacentes del vértice, pero en el caso de las aristas entrantes, deberemos eliminar el vértice en las listas de adyacentes de todos los demás vértices. Para cubrir el hueco dejado por el vértice eliminado, realizaremos un desplazamiento de la información hacia índices anteriores en los vectores *vértice* y *lista-ad.*

```
proc elim-vértice(e v : vértice, g : grafo)
  (b.p) := buscar(g.vértice, g.tamaño, v)
  si b entonces
    para i = p + 1 hasta g.tamaño hacer
      g.vértice[i — 1] := g.vérríce[í]
      g.lista-ad[i — 1] := g.lista-ad[i]
    fpara
    g.tamaño := g.tamaño — 1
    para i = 1 hasta g.tamaño hacer
      eliminaría. g ,lista-ad[i])
    fpara
  fsi
fproc
```

La operación para eliminar un vértice de una lista está implementada en el

Ejercicio 5.5.

Para eliminar una arista se averigua el índice del origen y se elimina el destino de la correspondiente lista de adyacentes.

proc eiim-arista(e u, *w : vértice, g : grafo) { (-)(g.tamaño))*

(b.p) = buscartg.*vértice, g.tamaño.* u)

si *b* **entonces** eliminar(u.', *g.lista-ad[p])* **fsi**

fproc

Para determinar si un vértice está en el grafo recurrimos una vez más a la función buscar.

fun esta-vértice?(u : *vértice, g : grafo)* **dev** *b : bool (O (g.tamaño))*

(b.p) := buscarfg. *vértice, g.tamaño, v)*

ffun

Para determinar si una arista está en el grafo se averigua el índice del origen y se busca el destino en la correspondiente lista de adyacentes.

fun está-arista?(u, *w . vértice, g : grafo)* **dev** *b : bool { 0(g.tamaño))*

(b.p) := buscar (g. vértice. g.tamaño, v)

b := b está?(w, g.lista-ad(p])

ffun

El grafo es vacío si el tamaño es 0.

fun es-grafo-vacío?(g : *grafo)* **dev** *b : bool* (0(1))

b := (g.tamaño = 0) **ffun**

Para obtener el conjunto de vértices adyacentes a uno dado, debemos primero localizar el vértice para después recorrer la lista correspondiente. Si el vértice no

estuviera en el grato, se produciría un error.

Como ya justificamos en la representación de los gratos mediante la matriz de adyacencia del Ejercicio 9.3, vamos a implementar la operación adyacentes para que devuelva una *lista* de vértices en lugar de un conjunto. De esta forma simplemente necesitamos copiar la lista de adyacentes del vértice dado.

fun adyacentes(i) : *vértice, g : grato)*
dcv / : *lista[vértice] (&(g.tamaño))*
(b. p) := buscar(g.vértice. *g.tamaño,* u)
si *b* **entonces**
I := copiar-lista(*g.lista-ad[/>])*
si no error(Vértice desconocido)
fsi
ffun

Todas estas operaciones (salvo grafo-vacío y es-grafo-vacío?) realizan llamadas a la función buscar, cuyo coste es lineal con respecto al número de vértices en el grafo. Por otra parte, estas operaciones también dependen del coste de las operaciones sobre las listas. Como una representación estática de las listas (con un tamaño máximo prefijado) no resolvería los problemas de espacio de la matriz de adyacencia, consideraremos para las listas una representación dinámica como la dada en el Ejercicio 5.3. con coste constante para las operaciones básicas de acceso a las listas. En realidad, en dicha representación se utilizan estructuras doblemente enlazadas, para permitir un acceso con coste constante por ambos extremos de las listas, pero para las listas de adyacentes basta con acceso eficiente por el extremo izquierdo, lo que permitiría una representación con estructuras de enlace simple. En cuanto a las operaciones de copiar una lista y buscar o eliminar un elemento

en una lista, tienen un coste lineal con respecto al tamaño de las listas (que está acotado por el número de vértices en el grafo).

De esta forma el coste de las operaciones añ-vértice. añ-arista, elim-arista. está-vertice?. esta-ansta? y adyacentes resulta ser lineal con respecto al número de vértices, mientras que el coste de la operación efim-vértice está en 0(n + *tn).* siendo *n* el número de vértices y *m* el número de aristas, pues recorre por completo todas las listas de adyacentes.

Si el conjunto de vértices está fijado desde el principio y la naturaleza de los vértices es irrelevante, la representación puede reducirse al vector de listas (de adyacentes). En estas condiciones las operaciones añ-arista. elim-arista, está-arista? y adyacentes tendrán un coste lineal respecto al tamaño de la lista de adyacentes, que está entre 0 y n; el resto de las operaciones sobre vértices no se consideran (véase la implementación de grafos valorados del Ejercicio 9.6).

En cuanto al coste en espacio de esta representación, está en *&(N + in),* o en (-)(*/*; + *m*) si lijamos en *n* el número de vértices del grafo *(m* continúa siendo el número de aristas).

Completamos una vez más la representación con una operación para *copiar* grafos (véase la Sección 2.1.3).

Aunque la base de la representación es un registro con tres campos, no se puede simplemente realizar una asignación de los campos del registro, porque el primer campo es un vector de listas y, como acabamos de comentar, posiblemente estén implementadas como estructuras enlazadas. Se requiere por tanto utilizar explícitamente la operación de copiar listas. El algoritmo queda como sigue

fun copiar-grafo(t? : *grato)* dev *f . grafo (Qfg.tamaño + m) }*

f.tamaño := g.tamaño

f.vértice[\..f.tamaño] := g.vértice[\ ..g .tamaño]

para *i* = 1 **hasta** *g.tamaño* **hacer**

f.lista-ad[i] := copiar-lista(g*.lista-ad]i])*

fpara ffun

Suponiendo que el coste de copiar las listas es lineal con respecto al tamaño de las mismas, la copia total de todas las listas de adyacentes será lineal con respecto a *m.* el número de aristas. La copia del vector *vértice* es lineal con respecto a su tamaño, de ahí que el coste total de copiar el grafo esté en *Q (g.tamaño + m).*

Consideramos ahora que el grafo ya tiene fijo el número de vértices *(ti).* Si comparamos esta representación con la de la matriz de adyacencia del Ejercicio 9.3, podemos afirmar que las listas de adyacentes son mucho menos

costosas en espacio cuando el grafo tiene pocas aristas, porque la matriz de adyacencia será una matriz *dispersa* con la mayoría de las entradas con valor falso. En cambio, cuando *m* se acerca
a *ir.* el coste en espacio de ambas representaciones es del mismo orden, cuadrático. pero la matriz de adyacencia es más sencilla y no necesita enlaces.

Con respecto al coste en tiempo, observamos que las operaciones individuales sobre aristas son más costosas con las listas de adyacentes, puesto que con la matriz se garantizan costes constantes. Sin embargo, si el algoritmo necesita determinar el conjunto total de aristas, o recorrer todas ellas de alguna manera, la matriz de adyacencia requerirá un coste en $0(n^2)$. mientras que las listas de adyacentes garantizarán un coste en 0(n -t-m). Cuando en el grafo hay pocas aristas, es decir, m es comparable ano menor, entonces el coste de recorrer todas las aristas con las listas de adyacentes estará en 0(n). claramente mejor que con la matriz de adyacencia. Ahora bien, si hay muchas aristas en el grafo y *ni* se acerca a *ir.* entonces la matriz de adyacencia ofrece operaciones más simples y con constantes multiplicativas más pequeñas.

Resumiendo, las listas de adyacentes son más eficientes cuando se tienen *grafos dispersos* (con pocas aristas), mientras que la matriz de adyacencia es

mejor opción cuando se trata de *grafos densos* (con muchas aristas).

En la representación que acabamos de detallar, las listas de adyacentes contienen los destinos de las aristas "de salida" de cada vértice. En algunas aplicaciones es necesario determinar los orígenes de las aristas de "entrada" de los vértices, y esto resulta muy costoso si solamente se dispone de las listas descritas. Por eso en ocasiones se utiliza una representación donde a cada vértice se le asocian dos listas, una "de entrada" y otra "de salida". Aunque se duplica el coste en espacio (la información se duplica), se reduce el coste en tiempo para determinadas consultas sobre el grafo. Nótese que las matrices de adyacencia no presentan esta asimetría, porque las filas se corresponden con las listas de salida y las columnas con las de entrada.

Apartado (b)----------------

En el caso de los grafos no dirigidos el tipo representante no cambia, aunque ahora al añadir/eliminar una arista (*v. ir*) hay que añadir/eliminar también la arista (u>. v).

proc añ-arista(e r. *ir : vértice, g : grafo)*

:= buscar(g.vértice. g.taniaño, v)

(/??. pz) '■= buscaría, *vértice, g. tamaño, w)*

si -'/q v **entonces** error(Vértices desconocidos)

si no

```
    si -^está?(ui. g.lista-ad[pi |)
entonces añadir-izq(u). g,/Ár«-o&lt;/[pi])
fsi
    si —'está^(i>. g.lista-ad[pz])
entonces añadir-izq(i',g.lista-ad[pz]) fsi
  fsi
fproc
proc ehm-arista(r. w : vértice, g : grafo)
  (b\. p\) := buscarfg.vértice. g.taniaño,
t>)
  \hz.pz) •= buscaríg.vértice, g.taniaño,
w)
  si /q A Ir entonces
    eliminar(u.'. g.lista-ad[p\])
    eliminar(t>. g.lista-ad[pz 1)
  fsi fproc
```

Las restantes operaciones se implcmentan igual que para los gratos dirigidos, y las consideraciones sobre los costes son igualmente aplicables.

Especificar un TAD para describir los *grafos valorados* con valores pertenecientes a un tipo dado como parámetro, con las siguientes operaciones:

- crear el grafo vacío,
- añadir un vértice,
- añadir una arista valorada,
- obtener el valor asociado a una arista,
- eliminar un vértice y todas las aristas que entran o salen de él,
- eliminar una arista,
- determinar si un vértice pertenece a un grafo,
- determinar si una arista pertenece a un grafo,
- determinar si un grafo es vacío, y
- obtener el conjunto de vértices adyacentes a uno dado.

(a) Especificar los *grafos valorados dirigidos.*

(b) Especificar los *grafos valorados no dirigidos.*

-----------------Solución-----------------

Las diferencias entre grafos valorados dirigidos y grafos valorados no dirigidos son las mismas que entre grafos dirigidos y no dirigidos que se detallan en la solución del Ejercicio 9.1. Así pues, nos limitaremos a desarrollar la especificación para los *grafos valorados dirigidos.*

La especificación depende de dos parámetros: el conjunto de vértices y el conjunto de valores para las aristas. Para el primero utilizaremos el parámetro

VÉRTICES, definido en el Ejercicio 9.1 y que requiere una relación de igualdad entre sus elementos; el segundo no precisa ninguna condición especial y lo definimos como sigue:

parámetro *VALORES*
tipos *valor*
fparámetro

que es un renombramiento del parámetro *ELEM.* dado en la Sección 1.1.5.

El conjunto de constructoras es básicamente el mismo que para los grafos no valorados: crear un grafo vacío, añadir un vértice y añadir una arista con un valor asociado (esta última sigue siendo parcial). También las ecuaciones de equivalencia son prácticamente las mismas, pero ahora añadir una arista que ya estaba incluida tiene el efecto de actualizar con el nuevo valor su valor asociado. Por tanto, las constructoras *no son libres.*

La operación de consulta que devuelve el valor asociado a una arista solo tiene sentido si la arista está en el grafo, por lo que se define como parcial.

especificación *GRAFOS-VALORADOS[VÉRTICES, VALORES]*
usa *BOOLEANOS, CONJUNTOS[VÉRTICES]*
tipos *grafo-val*
operaciones
gv-vacío : —>*grafo-val* (constructora }
gv-añ-vértice : *vértice grafo-val* —> *grafo-val* { constructora }

gv-añ-arista *: vértice vértice valor grafo-val —rp grafo-val*
(constructora) gv-valor *: vértice vértice grafo-val —> p valor*
gv-elim-vértice: *vértice grafo-val —> grafo-val*
gv-elim-arista : *vértice vértice grafo-val —> grafo-val*
gv-está-vértice?*vértice grafo-val —» bool*
gv-está-arista?*vértice vértice grafo-val —> bool*
gv-es-vacío? *grafo-val —» bool*
gv-adyacentes*vértice grafo-val —>„ conjunto[vértice]*

variables

v. tu, v'. w' : vértice
p.q. p' : valor
g : grafo-val

Recordemos que la operación gv-añ-arista da error si los vértices extremos no están en el grafo; respetando dicha restricción, los vértices y aristas se pueden añadir en cualquier orden. Añadir varias veces el mismo vértice no tiene efecto, pero si se añade varias veces la misma arista, el valor válido será el último, por lo que el orden de construcción es importante.

ecuaciones

gv-añ-arista(u, *w, p. g)* = error <= -■gv-está-vértice?(v. g) *v* -■gv-está-vértice? tu.'. g)
gv-añ-vértice(u. gv-añ-vérticeír, g)) = gv-añ-vértice(r. g)
gv-añ-vértice(ir, gv-añ-vértice(i). g))

=gv-añ-vértice(v. gv-añ-vértice(ir. g)>

gv-añ-arista(u. *w. p,* gv-añ-arista(i), *w. q.* g)) =gv-añ-arista(r. *w. p.* g)

gv-añ-arista(i’. *w, p,* gv-añ-arista(i)'. *w'. p'.* g)) = gv-añ-arista(u'. ir'. *p'.* gv-añ-arista(r. ir. *p.* g))

<= r i>' v m ? *w'*

gv-añ-arista(i', *w. p.* gv-añ-vértice(u', g)) = gv-añ-vértice(v'. gv-añ-arista(r. u’, *p.* g))

<= (1/ *v* A u' ir) v (gv-está-vértice?(v. g) A gv-está-vértice?(ir. g))

A partir de ahora consideraremos que los términos formados por constructoras que aparecen en las ecuaciones son correctos.

Según se ha comentado anteriormente, el valor de una arista es el correspondiente a la última vez que la arista fue añadida. Si no se ha añadido ninguna vez, se producirá un error.

gv-valor(t>, *w.* gv-vacío) = error

gv-valor(u, m,gv-añ-vértice(i)', g)) = gv-valor(r. ir. g)

gv-valor(i>. ir, gv-añ-arista(i>, *w, p.* g)) = *p*

gv-valor(r. ir, gv-añ-arista(u', *w'. p'.* g)) =gv-valor(i\ir, g) <= r 7= *v'* v ir ?= ir'

l.as ecuaciones para el resto de las operaciones modificadoras y observadoras son similares a las dadas para los grafos no valorados, puesto que el valor asociado a las aristas no es

relevante en estas operaciones. Véase la solución del Ejercicio 9.1 para más explicaciones.

gv-elim-vértice(u. gv-vacío) = gv-vacío

gv-elim-vértice(i'. gv-añ-vértice(i'. g)) = gv-elim-vértice(i'. *g)*

gv-elim-vértice(r, gv-añ-vértice(ic. g)) = gv-añ-vértice(ii', gv-elim-vérticeti'. g))

<= *v* # IT

gv-elim-vértice(u. gv-añ-arista(v, ir, *p.* g)) = gv-elim-vértice(r. g)

gv-elim-vérticeíi[1], gv-añ-ansta(ir, u, *p.* g)) = gv-elim-vértice(r, g)

gv-elim-vértice(u. gv-añ-arista(u'. *w',* *p'*, g)) = gv-añ-arista(i''. ir'. *p* . gv-elim-vértice(r, g))

<= i> ^if A v / ir'

gv-elim-arista(i>, *w.* gv-vacío) = gv-vacío

gv-elim-arista(u, *w.* gv-añ-vértice(r', g)) = gv-añ-vértice(v . gv-elim-arista(r. ir. g))

gv-elim-arista(u, *w.* gv-añ-arista(r, ir, *p,* g)) = gv-elim-ansta(r,ir, g)

gv-elim-arista(u, *iv.* gv-añ-arista(i>', tu', //, g)) = gv-añ-arista(r . ir , *p'.* gv-elim-arista(r, ir, g))

<= *v* r' v ir 7= ir'

gv-está-vértice?(u. gv-vacío) = falso

gv-está-vértice?(r, gv-añ-vértice(u», *g))*

= r == ir V gv-está-vértice?tr. g)

gv-está-vértice?(u, gv-añ-arista(r', *u<',* *p'*. g)) = gv-está-vértice?(r, g)

gv-está-arista?(v, *w.* gv-vacío) = falso
gv-está-arista?(i', *w.* gv-añ-vértice(i>',
g)) = gv-está-arista?(v. *w. g)*
gv-está-arista?(i>. *tu.* gv-añ-arista(i)',
w'. p'. g)) = *(v == v'* A *w ==* w')
v gv-está-arista?(i). *w.*
g)
gv-es-vacío? (gv-vacío) = cierto
gv-es-vacío?(gv-añ-vértice(v. g)) =
falso
gv-es-vacío?(gv-añ-arista(v, *w. p,* g)) =
falso
gv-adyacentes(i). g) = error <= -■gv-
está-vértice?(i». g)
gv-adyacentes(v, gv-añ-vértice(i>. *g))*
= cjto-vacío <= -■gv-está-
vértice?(u, g)
gv-adyacentes(i), gv-añ-vértice(u). g))
= gv-adyacentes(i>, *g)*
<= *v w V* gv-está-
vértice?(u. g) gv-adyacentes(i), gv-
añ-arista(i>. *w. p,g))* =
añadir(ur. gv-adyacentes(i), g))
gv-adyacentes(i>. gv-añ-arista(t)', *w'.*
p'. g)) = gv-adyacentes(v, *g)* <=
v/v'

fespecificación

Diseñar una representación del TAD de los grafos valorados dirigidos suponiendo que el conjunto de vértices del grafo está prefijado y los vértices están representados con naturales 1 *..n.* Implemen- **tar** las operaciones para crear un grafo sin aristas, añadir y eliminar aristas, determinar si existe una arista, obtener el valor asociado a una arista y

obtener el conjunto de vértices adyacentes a uno dado, especificadas en el Ejercicio 9.5.

(a) Utilizar una *matriz de valores* que para cada par de vértices indique el valor asociado a la arista correspondiente (si esta existe).

(b) Utilizar *listas de adyacentes,* donde para cada vértice en el grafo se tiene una lista con todos sus vértices adyacentes, cada uno de ellos junto con el valor de la arista correspondiente.

------------------Solución-----------------

Apartado (a)

Siguiendo las ideas expuestas en el Ejercicio 9.3 para la representación de grafos no valorados, representamos la matriz de valores mediante un vector bidimensional cuyos elementos tienen tipo *valor* y cuyos índices son los números naturales l.ji. Para que la matriz de valores esté definida en todas sus entradas, elegiremos un valor especial, v-especial, para los casos en que la arista no exista en el grafo.

Por ejemplo, al grafo de la Figura 9.1 (c) (suponiendo que los vértices se numeran en el orden alfabético *a. fi, y. S)* le corresponde la matriz de valores siguiente:

oo 37 oo
oo*oo* I 5
oooo oo 1
24oo oo

donde hemos representado v-especial mediante oo.

El tipo representante es el siguiente:

tipos

grafo-val[n] **= vector! 1 ■». 1-nJ de** *valor*

ftipos

Cuando el grafo no tiene aristas, todas las entradas de la matriz tienen el valor v-especial.

fun gv-sin-aristas() **dev** *g : grafo-val[n]*
(0(rr))
 g[l..n, I../1] := [v-especial]
ffun

Para añadir una arista basta asignar el valor asociado a la entrada correspondiente en la matriz de valores. Si la arista ya tenía un valor previo asociado, simplemente queda actualizado con el nuevo valor.

proc gv-añ-arista(e v, *tu* : I *..n.* **e** *ni : valor, g : grafo-val[n])* {0(1)}
 g[u. *u*>| *:= ni*
fproc

Para eliminar una arista se asigna el valor especial a la entrada correspondiente en la matriz.

proc gv-elim-arista(e *v. w* **: l..n.g :** *grafo-val[n])* **{ 0(1))**
 g[u, m| := v-especial
fproc

Para comprobar si una arista está en el grafo basta comprobar que la correspondiente posición de la matriz de valores no contiene el valor especial.

fun gv-está-arista?(u. *w* : l..n. g : *grafo-val[n]*) **dev** *b : bool* **{ 0(1))**
 b := g[u, tu] v-especial **ffun**

Para determinar el valor asociado a una arista hay que consultarla entrada correspondiente en la matriz de valores si esta contiene el valor especial significa que la arista no existe en el grafo y se producirá un error.

fun gv-valor(i’, *w* **: 1** *n.g : grafo-val[n])* dev *ni : valor* (0(1) |

si g[r. u> | = v-especial **entonces** error(La arista no existe)
si no *ni* := g[u. tu]
fsi
ffun

Para obtener los vértices adyacentes a un vértice *v* se recorre la fila c de la matriz. Para aquellas entradas que no sean el valor especial, se añade el número de columna al conjunto de adyacentes de r. Como ya hemos justificado anteriormente para los grafos no valorados (en el Ejercicio 9.3). obtenemos una *lisia* de vértices en lugar de un conjunto.

fun gv-adyacentes(u : 1.,/r.g : *gralo-val[n])* **dev** *l : lista[l..n]* **{ 0(«))**
/ := lista-vacía()
para *i* **= 1 hasta** *n* **hacer**
si g[u.í| *r* v-especial entonces añadir-der(/. í) fsi
fpara
ffun

Apartado (b)-------------

Siguiendo las ideas expuestas en el Ejercicio 9.4 para la representación de grafos no valorados, las listas de adyacentes se organizan en un vector de listas indexado por 1..*n*. En este caso, las listas contendrán pares de la forma (*destino, valor*), indicando el vértice destino de la arista y su valor asociado. Por tanto, el tipo representante es el siguiente:

tipos
info-arísta **= reg**
destino : l..n
valor : valor
freg
grafo-val = **vectorfl..**M] **de** *lista[info-aristal*
ftipos

Cuando el grafo no tiene aristas, todas las listas de adyacentes son vacías.

fun gv-sin-aristas() **dev** *g : grafo-val[n]*
(0(n))
g[l..n] := [lista-vacía()]
ffun

Al añadir una arista hay que tener en cuenta si dicha arista ya existe en el grafo, porque en ese caso debemos sustituir el valor antiguo por el nuevo.

proc gv-añ-arista(e *v. w* : l..n, **e** m : *valor, g : grafo-val[n]}* (0(zi) } **var** *l : lista[info-arista], a : info-arísta*
b := falso
l g[v] : g[v] '■= lista-vacía()
mientras -’es-lista-vacía?(/) hacer
a := izquierdo(Z) ; elim-izq(Z)

si *w* = *a.destino* **entonces** *b* := cierto ; *a.valor* := *m* **fsi**
añadir-der(g[u], *a)*
fmientras
si -•*b* **entonces** (la arista es nueva)
a.destino := *w* ; *a.valor* := *ni*
añadir-der(g[v]. *a)*
fsi
fproc

Para eliminar una arista, hay que eliminar el vértice destino de la lista correspondiente al vértice origen.

proc gv-elim-arista(e u, *w* : l..n,g : *grafo-val[n])*
eliminar(iu. g[u])
fproc

Debemos tener en cuenta que la operación que elimina un elemento de una lista implementada en el Ejercicio 5.5 tiene que recibir como argumento el par completo, en vez de solamente la primera componente del par, por lo que habría que adaptar la implementación para usar esa operación de la forma adecuada en el procedimiento anterior.

Para determinar si una arista está en el grafo hay que buscar el destino en la lista de adyacentes del origen.

fun gv-está-arista?(v, *w* : l.ji, *g : grafo-val[n])* dev *b : bool* **var / :** *lista[info-arista], a : info-arísta*
b := falso
l := copiar-lista(g[u]J
mientras -'Zt A -■es-lista-vacía?(/) **hacer**
a := izquierdo(/) : elim-izq(Z)

```
    si w = a.destino entonces b := cierto
fsi
  fmientras
  ffun
```

Para determinar el valor asociado a una arista se busca el destino en la correspondiente lista de adyacentes; el valor que lo acompaña será el valor de la arista.

fun gv-valor(v, *w : l..n.g : grafo-val[n])*
dev *m : valor* **var** *I : lista[info-aristal. a : info-ansta*
b := falso
I := copiar-lista(g[v))
mientras --*b* A -’es-lista-vacía[7](/)
hacer
a := izquierdo!/) ; elim-izq(Z)
si *w = a.destino* **entonces** *b :*= cierto
; *m := a.valor* **fsi**
(mientras
si -•*b* **entonces** error(La arista no existe) **fsi ffun**

La lista de adyacentes a un vértice *u* se obtiene copiando todos los destinos de la lista de adyacentes de *u.*

fun gv-adyacentes(u : *l..n.g : grafo-val[n])* **dev / :** *lista[l..n]*
var /' *lista[info-arista]. a : info-arista*
/ := lista-vacíaf)
/' .= copiar-lista(g[u])
mientras -’es-lista-vacía?(/') hacer
n = izquierdo!/') ; elim-izq(Z')
añadir-der(/, *a.destino)*
(mientras
ffun

El coste de todas estas operaciones es lineal con respecto al tamaño de la lista de adyacentes sobre la que se trabaja, el cual está acotado por *n.* el número de vértices.

Se puede conseguir un coste constante para la operación de añadir una arista si en lugar de comprobar si dicha arista ya existe, nos limitamos a añadir la arista con el nuevo valor por la izquierda de la lista, porque al consultar el valor de una arista siempre se encontrará primero el más reciente. El problema con esta opción es que ya no es posible acotar el tamaño de las listas de adyacentes, las cuales se van llenando con pares inútiles que hacen más costosas las otras operaciones.

Extender el TAD de los grafos con una operación que obtenga el conjunto de las aristas **de un grafo.**
(a) Implementarla para la representación con matriz de adyacencia.
(b) Implementarla para la representación con listas de adyacentes.

------------------Solución**----------------**
Empezamos definiendo varias especificaciones para construir aristas según el tipo de grato que nos interese.
Para grafos dirigidos una arista se identifica con un par ordenado de vértices.
especificación *AFIISTAS[VÉRTICESi*
tipos *arista*
operaciones
(_, _) : vértice vértice —» arista { constructora)
fespecificación
Para grafos no dirigidos una arista

consiste en un par no ordenado de vértices, lo cual se consigue imponiendo una ecuación de equivalencia.

especificación *ARISTAS-ND[VÉRTICES]*
tipos *arista-nd*
operaciones
(: vértice vértice —> arista-nd { constructora)
variables
v.w : vértice
ecuaciones
(v.w) = (w.v)
fespecificación

Para grafos valorados (dirigidos) cada arista lleva asociado un valor.

especificación *ARISTAS-VALORADAS[VÉRTICES, VALORES]*
tipos *arista-val*
operaciones
: vértice vértice valor —> arista-val (constructora)
fespecificación

El comportamiento de la operación que obtiene el conjunto de aristas de un grafo se especifica por distinción de casos sobre las tres constructoras de los grafos. En el caso de grafo-vacío no hay aristas y el conjunto es vacío: obviamente, añadir nuevos vértices con añ-vértice no afecta al conjunto de aristas: y al añadir una nueva arista con añ-arista esta se añade también al conjunto.

especificación *GRAFOS-ARISTAS[VÉRTICES]*
usa *GRAFOS[VÉRTICES], CONJUNTOS[ARISTAS[VÉRTICES]]*

operaciones
aristas : *grafo —> conjunto[arista]*
variables
v, *w : vértice*
g : grato
ecuaciones
aristas(grafo-vacío) = cjto-vacío
aristas(añ-vértice(u, g))= aristas(g) '
aristas(añ-arista(v, *w. g))* = añadir((v, *w*). aristas(g))
fespecificación

Cuando el grafo es no dirigido, la idea es la misma pero usando el correspondiente tipo de aristas, *arista-nd* en vez de *arista,* por lo que no la repetimos.

En el caso de grafos valorados (dirigidos) usamos el tipo *arista-val.* Además, si se añade la misma arista varias veces, hay que quedarse con el último valor. Lo más sencillo es eliminar del grafo todas las adiciones previas.

especificación *GRAFOS-VALORADOS-ARISTAS[VÉRTICES. VALORES]*
usa *GRAFOS-VALORADOS[VÉRTICES, VALORES],*
CONJUNTOS[ARISTAS-VALORADAS[VÉRTICES, VALORES]]
operaciones
aristas : *grafo-val —> conjunto[arista-vali* **variables**
v, w : vértice
p : valor
g : *grafo-val*
ecuaciones
aristas(gv-vacío) = cjto-vacío
aristas(gv-añ-vértice(i'. g))= aristas(g)

aristas(añ-arista(*v.w.p.g))* = añadir((v. u>. p), aristastgv-elim-anstair. u . g)))
fespecificación

Como ocurre con los conjuntos de vértices adyacentes, la aplicación práctica de esta operación aristas es poder recorrer todas las aristas de un grafo. Por eso vamos a implementar esta operación de forma que devuelva una *lista* con las aristas en lugar de obtener un conjunto.

Apartado (a)----------------

En el caso de la matriz, de adyacencia hay que recorrer todas las filas de la matriz, añadiendo a la lista aquellas entradas que tengan el valor cierto.

tipos
 arista = **reg**
 origen : vértice
 destino : vértice
 freg
ftipos
fun aristas-matrizfg : *grato}* **dev / :** *lista[arista] (&(g.tamaño-))* **var** *a arista*
 I := lista-vaciaO
 para *i* = I **hasta** *g.tamaño* **hacer**
 para *j* = 1 **hasta** *g.tamaño* **hacer**
 si *g maíriz[i,* **j] entonces**
 a.origen := vértice[i] : a.destino := vértice!/]
 añadir-der(i. *a)*
 fsi
 fpara
 fpara
ffun

En el caso de grafos no dirigidos, este

algoritmo obtendría una lista con "el doble" de aristas, ya que se incluyen en los dos sentidos. Es fácil reducir la lista y que cada arista aparezca solamente en un sentido, en concreto con origen en el vértice que se incluyó antes en el grafo. Para ello basta recorrer cada tila solamente a partir de los vértices posteriores (incluido él mismo por si hubiera auto-aristas), es decir, en vez de recorrer toda la matriz se recorre la mitad superior (incluida la diagonal principal).

fun nd-aristas-matriz(g : *grato)* **dev / :** *lista[aristai { Q(g.tamaño-))* **var** *a . arista*

I := lista-vaciaO

para *i* **= 1 hasta** *g.tamaño* **hacer**

para *j = i* **hasta** *g.tamaño* **hacer**

si *g.inatriz[i. j* **entonces**

a.origen := i’értice[i| ; *a.destino := vértiee[*j)

añadir-der(/. *a)*

fsi

fpara

fpara

ffun

Cuando el grafo es valorado, se añaden a la lista las entradas de la matriz de valores que no tengan el valor especial que indica la no existencia de la arista.

tipos
arista-val = **reg**
origen : vértice
destino : vértice
valor: valor
freg
ftipos
fun gv-aristas-matriz(g : *grafo-val)* dev *I : Hsta[arista-val] { ©(g.tamaño^2))* **var** *a : arista-val*
I := lista-vacía()
para *i* — 1 **hasta** *g.tamaño* **hacer**
para *j* = 1 **hasta** *g.tamaño* **hacer**
si *g.matriz[i. j]* / v-especial entonces
a.origen := vértice[i] ; a.destino := vértice[j]
a.valor g.matriz[i, j]
añadir-der(/, *a)*
fsi
fpara
fpara
ffun

Suponiendo que la operación de añadir un elemento a una lista tiene un coste constante, todos estos algoritmos tienen un coste cuadrático con respecto al número de vértices del grafo.

Apartado (b)-------------------

Cuando el grafo está representado por listas de adyacentes, hay que recorrer cada una de las listas.

fun arístas-listasfg : *grato)* dev / : *lista[arista]*
var /' : *Hsta[vértice], a : arista*
  l := lista-vacía()
  para *i* = I **hasta** *g.tamaño* **hacer**
    l[1] := copiar-lista(*g.lista-ad[i])*
    mientras -'es-lista-vacía?(/') **hacer**
      v := izquierdo(Z') ; elim-izq(/')
      a.origen := vértice[i] : a.destino := v
      añadir-der(/, *a)*
    fmientras
  fpara
ffun

El coste de este algoritmo está en *0(n + m),* siendo *n* el número de vértices y *m* el de aristas.

Para grafos no dirigidos tendremos el siguiente algoritmo, que obtiene un conjunto "reducido" donde para cada arista el origen no es posterior al destino.

fun nd-aristas-listas(g : *grafo)* **dev / :** *lista[arista]*
var /' : *lista[vértice]. a : arista*
  l lista-vacía()
    para *i* = 1 **hasta** *g.tamaño* **hacer**

/' := copiar-lista(g./ísíí/-nd[í])
mientras -'es-lista-vacía?(/) **hacer**
v := izquierdo(Z') ; elim-izq(/')
(b.p) := buscar(g.vértice, g.tamaño, v) (b = cierto |
si *i* < *p* **entonces**
a.origen := vérrZcefí] ; *a.destino :=* **r**
añadir-deríZ. *a)*
fsi
(mientras
fpara
ffiin

El coste ahora está en *(r)(nm).* siendo *n* el número de vértices *y m* el de aristas, porque para cada arista del grafo hay que realizar una búsqueda de su destino en el vector de vértices. Como ya se comentó cuando se detallaron las representaciones de los gratos, en la práctica los vértices se representan mediante números naturales y no es necesario realizar estas costosas búsquedas, con lo que el coste de esta versión sería similar al de la versión para gratos dirigidos.

En el caso de gratos valorados supondremos que las listas de adyacentes solamente contienen el último valor para cada arista. Como el algoritmo queda muy similar al dado para los gratos no valorados, no lo detallamos aquí.

Diseñar un algoritmo que. dado un grafo G y un vértice inicial v, recorra *en profundidad* todos los vértices en G accesibles desde *v.*

Solución:

El recorrido en *profundidad* (en inglés *Depth First Search.* abreviado DFS) de un grafo es equivalente al recorrido en profundidad (en preorden) en los árboles (véanse los Ejercicios 6.9 y 6.17). La diferencia es que en los árboles no hay ciclos y siempre se llega a una hoja antes de iniciar el "retroceso". En el caso de los gratos, hay que detectar si el vértice al que se llega ha sido ya visitado. De esta forma, el recorrido parte del vértice inicial y continúa por alguno de sus adyacentes, recorriendo un camino hasta llegar a un vértice previamente visitado o hasta que el camino queda truncado porque no hay adyacentes en el vértice alcanzado. Llegados a ese punto, se retrocede por el camino hasta el primer vértice que ofrezca algún adyacente válido (no visitado) por el que poder continuar.

Por ejemplo, para el grafo de la Figura 9.2. un posible recorrido en profundidad desde el vértice I visitaría los vértices en el orden siguiente: I. 2. 3. 5. 6. 8. 7.4; el resto de los vértices (9. 10. II. 12) no son visitados porque no son accesibles desde el vértice I.

La forma más sencilla de implementar este recorrido es mediante un algoritmo recursivo. Además de "visitar" cada vértice, el algoritmo asignará a cada vértice un número natural que indique la posición en la que ha sido recorrido. De esta forma, en lugar de llevar un vector de valores booleanos para marcar los vértices visitados, la posición de cada

vértice no visitado será 0.
proc DFS(e G *: grafo[n]. e v :* **1** *n. k : 0..n. recorrido[***l../i] de 0..n) var / :** *lista[] n]*
 k := k + 1 : *recorrido[t*> 1 *:= k :* visitar(v)
 / := adyacentes(u. G)
 mientras -’es-lista-vacía?(Z) **hacer**
  w := izquierdo(Z) ; elim-izq(Z)
  si *recorrido[w]* = 0 **entonces** DFS(G. ir. *k. recorrido')* **fsi**
 fmientras
fproc

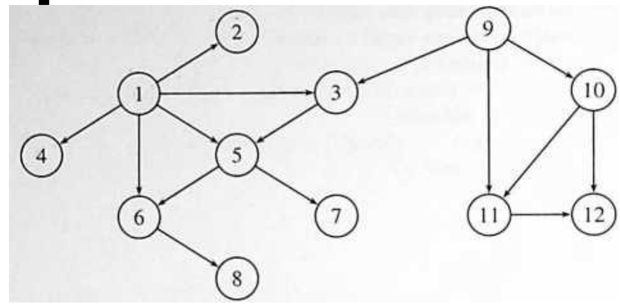


Figura 9.2: Ejemplo de grafo dirigido.

La llamada inicial se realiza con el vértice inicial, con *k* = 0 y el vector *recorrido* inicializado a 0 en todas sus componentes. Nótese que al final *k* indica el número de vértices recorridos.

En realidad, el algoritmo está parametrizado con respecto a la operación visitar que. dependiendo de la aplicación del recorrido, realizará unas acciones u otras con el vértice de turno. Además, es válido tanto para grafos dirigidos como no dirigidos (y para grafos valorados).

Este algoritmo puede utilizarse para recorrer *todos* los vértices de un grafo de la forma siguiente: se pane de un vértice

cualquiera y se aplica el procedimiento DFS: al finalizar se escoge otro vértice cualquiera que todavía no haya sido visitado y se vuelve a invocar DFS. y así sucesivamente hasta que ningún vértice del grafo tenga su recorrido a 0. Se puede guiar el recorrido imponiendo el orden en el que se desea probar cada vértice.

proc DFS-completo(e G : *grafo[n],* e *orden*\ 1../1] de 1 ../i, *recorrido*\ 1.,*n*] de 0../I)
 k := 0 : *recorrido*[l..n] := [0]
 para *i* = 1 **hasta** *n* **hacer**
  u *:= orden[i]*
  si *recorrido[v]* = 0 entonces DFS(G, o. *k, recorrido)* fsi
 fpara
fproc

La obtención de los vértices adyacentes depende de la representación escogida para el grafo. Con la representación de listas de adyacentes (véase el Ejercicio 9.4) se consigue que el coste del recorrido en profundidad del grafo completo esté en C-)(n + *in)* (siendo *m* el número de aristas en el grafo) porque cada vértice se visita exactamente una vez y el trabajo para el recorrido de adyacentes es proporcional al número de aristas. En cambio, si se utiliza la matriz de adyacencia (véase el Ejercicio 9.3). el coste resulta estar en 0(/r), porque se recorren todas las filas de la matriz.

Cuando el grafo es no dirigido y conexo, el recorrido en profundidad del grafo

desde cualquier vértice inicial genera (implícitamente) un *árbol general* (véase el Ejercicio 6.14) que contiene a todos los vértices del grafo y cuya raíz es el primer vértice visitado. Si nos olvidamos de cuál vértice es la raíz y del orden entre los hijos, obtenemos un *árbol (libre) de recubrimiento,* que es un subgrafo acíclico conexo que contiene lodos los vértices del grafo recorrido.

Se puede obtener una versión iterativa del recorrido en profundidad. Para ello se necesita una estructura donde almacenar el trabajo que se va quedando “pendiente", es decir, los vértices que hemos visitado pero de los cuales todavía nos quedan vértices adyacentes por explorar. Partí satisfacer la condición de recorrido en profundidad, estos vértices deben ser atendidos en orden inverso al de su incorporación a la estructura; por tanto, se necesita una estructura LIFO o pila (véase el Capítulo 3). En la pila cada vértice se almacena junto con la lista de sus adyacentes que quedan por explorar, la cual no puede ser modificada

sin desapilar el vértice; por eso debemos desapilar y apilar de nuevo el vértice en la cima cada vez que se explora uno de sus adyacentes.

El algoritmo iterativo para el recorrido en profundidad (desde un vértice) queda como sigue:

tipos
info-vértice = **reg**
vértice : l..n
adyacentes : lista[\..n]
freg
ftipos
proc DFS-it(e G : *grafo[n].e* u : l..n,recorrir/o[l..n] **deO../t)**
var *P : pila[info-vérticel. p. q : info-vértice*
k := 1 ; *recorrido[* 1. .n] := [0]
P **:= pila-vacía()**
recorrido[v] := 1 ; visitar(v)
p.vértice := v ; p.adyacentes := adyacentes!t>. G)
apilar!*p. P)*
mientras -■es-vacía?(*P***) hacer**
p cima(P) ; desapilar! P)
si ^es-lista-vacía?(p.a</yncewes) **entonces**
ir := izquierdo(p.adyacentes) ;
elim-izq(*p.adyacentes) :* apilar!*p. P)*
si recomdofut] = 0 **entonces**
A *:= k:* + 1 ; *recorrido[w] := k ;*
visitar(u.’)
q.vértice := w ; q.adyacentes := adyacentes(u’, G)
apilar(c/. *P)*
(si

(si

(mientras

fproc

En realidad, lo que se ha hecho es expltcitar el manejo de la pila que el mecanismo de reeursión utiliza de forma transparente al programador. La profundidad máxima de la pila es *n.* puesto que cada vértice se apila como máximo una vez. Para cada elemento en la pila tenemos un coste extra en espacio para la lista, pero el tamaño total de todas las listas que pueda haber en la pila es *ni.* En resumen, el coste en espacio adicional está en W(n + *ni).*

Escribir un algoritmo para determinar si un grafo no dirigido es un árbol libre.

-------------------Solución-------------------

Un grafo no dirigido es un árbol libre si es conexo y es acíclico, o equivalentemente, si es conexo y tiene exactamente *n* — 1 aristas para *n* vértices. Por ejemplo, el grafo de la Figura 9.3(a) es un árbol libre.

Para determinar si un grafo es conexo basta realizar un recorrido (por ejemplo en profundidad) desde cualquier vértice y que el número de vértices recorridos sea exactamente n. En el algoritmo que sigue se utiliza DFS (véase el Ejercicio 9.8) para realizar el recorrido en profundidad desde el primer vértice.

fun conexo?(G *: grafo[n])* dev *b : bool*
var *recorridol* **1 . ./r] de 0..n**
k := 0 ; recorri</o[l..n] := [0)
DFS(G. I, *k, recorrido)*
b := (k = n)
ITun

(a) Árbol libre (b) Arborescencia (c) No es una arborescencia

Figura 9.3: Ejemplos de grafos.

Resulta sencillo modificar DFS para que además contabilice el número de aristas encontradas, se utilicen para el recorrido o no. Como el grafo es no dirigido, hay que tener cuidado para no contabilizar dos veces cada arista. Contaremos solamente las aristas cuando el origen sea menor o igual que el destino. Se tiene entonces el siguiente algoritmo:

proc DFS-aristas(e G : *grafo[n],e v : l..n,k,a* : O../t, *recorrido[\ ..n]* de O.JI)
var / : *lista[l..n]*
k := k + 1 ; *recorrido[v] := k*
i := adyacentes(i>, G)
mientras -■es-lista-vacía?(/) **hacer**
w := izquierdo(í) ; elim-izq(/)
si *v* < *w* **entonces** *a := a* + 1 **fsi**
si *recorrido[w]* = 0 **entonces** DFS-aristas(G. *w, k, a. recorrido)* **fsi**
fmientras
fproc

fun árbol-libre?(G : *grafo[n])* **dev** *b : bool*
var *recorrido[\..n]* **de** 0../1
k := 0 ; a := 0 ; *recorrido[\..n] :=* [0]
DFS-aristas(G, 1, *k, a, recorrido)*
b := (k = n r\ a = n — 1) **ffun**

El coste de estos algoritmos es del mismo orden que el del recorrido en profundidad, es decir, *Q(n+m)* para

listas de adyacentes y 0(/r) para la matriz de adyacencia.

Se dice que un grafo dirigido es una *arborescencia* si existe un vértice, llamado raíz, desde el que se puede alcanzar cualquier otro vértice a través de un camino único. Escribir un algoritmo para determinar si un grafo dirigido es una arborescencia y, en caso de serlo, determinar su raíz.

------------------Solución-----------------

El grafo dirigido de la Figura 9.3(b) es una arborescencia mientras que el grafo dirigido de la Figura 9.3(c) no lo es.

La diferencia entre las arborescencias y los árboles generales (véase el Ejercicio 6.14) es que en las primeras no existe un orden entre los hijos de cada nodo. Como ocurre en los árboles generales, en una arborescencia no puede haber ciclos porque entonces habría infinitos caminos entre los vértices del ciclo. En consecuencia, tiene que existir al menos un vértice con grado de entrada 0, pues de lo contrario, partiendo de cualquier vértice y recorriendo aristas en sentido inverso (del destino hacia el origen), generaríamos un camino infinito, lo cual implicaría la existencia de un ciclo en el grafo. Por otra parte, no puede existir más de un vértice con grado de entrada 0. porque esos vértices no serían alcanzables entre sí. Además, ese vértice único con grado de entrada 0 tiene que ser la raíz de la arborescencia.

De esta forma, para determinar la

(posible) raíz de la arborescencia hay que encontrar la columna en la matriz de adyacencia que tiene a falso todas sus entradas (coste en $0(n^2)$). o el vértice que no aparece en ninguna de las listas de adyacencia (coste en O(n + m)).

Una vez determinada la raíz basta con realizar un recorrido en profundidad desde dicha raíz. Si en algún momento se tropieza con un vértice ya visitado, significa que existe más de un camino desde la raíz a dicho vértice. Además, para ser arborescencia, el recorrido tiene que haber visitado todos los vértices.

El algoritmo queda como sigue:

proc DFS-arborescencia(e *G : grafo[n], e v* : l..n. *k* : 0..n. *visitado[\..n]* **de** *bool. arb : bool)* **var** *I : lista[\..n]*
k := k + 1 : *visitado[v]* := cierto
I := adyacentes(v, G)
mientras -'es-lista-vacía?(/) **hacer**
ui := izquierdo(Z) ; elim-izq(/)
si *visitado[w]* **entonces** *arb :=* falso
si **no** DFS-arborescencia(G, *ui. k, visitado, arb)*
fsi
(mientras
fproc
fun arborescencia?**(G :** *grafo[n])* **dev (***b : bool. raí-* : l..u)
var *grada-de-entrada[l..n]* **de** *nat, visitado[l..n]* **de** *bool*
grado-de-entrada := obtener-grados-de-entrada(G)
r := 0
para *i* = I **hasta** *n* **hacer**

si *grado-de-entrada[i]* = **0 entonces** *r := r* **+ 1 :** *raíz* '■= *i* **fsi**
fpara
si *r* = 1 **entonces**
k .= 0 ; *b* := cierto ; *visitado[l..n]* := [falso]
DFS-arborescencia(G, *raíz, k, visitado, b)*
b := (b A *k = n)*
si no *b* := falso
fsi
ffun

Para obtener el grado de entrada de todos los vértices hay que "recorrer" todas las aristas del grafo. La forma de hacerlo dependerá de la representación elegida para el grafo. En el caso de tener una matriz de adyacencia, el grado de entrada de un vértice se corresponde con el número de valores iguales a cierto de la columna correspondiente; así que, para obtener los grados de todos los vértices, hay que recorrer todas las columnas de la matriz, y el coste estará en *Q(ir)* (en el Ejercicio 14.3 se detalla un algoritmo para obtener el grado de entrada y de salida de los vértices de un grato a partir de su matriz de adyacencia). Si se tienen listas de adyacentes, hay que recorrer todas las listas de adyacentes y sumar 1 al grado de cada vértice que encontremos en dichas listas. El coste total estará en ®(n + *ni),* siendo *ni* el número de aristas.

De esta forma, y teniendo en cuenta las consideraciones hechas sobre el coste del

recorrido en profundidad. se deduce que el coste de arborescencia? resulta estar en 0(n + *ni*) para listas de adyacentes y en *P)(>r)* para la matriz de adyacencia.

Demostrar que todo grafo no dirigido conexo contiene un vértice que puede ser eliminado de forma **que el grafo** sigue siendo conexo. Diseñar un algoritmo que, dado un grafo conexo $G = (V, A)$, encuentre dicho vértice con un coste en tiempo en $O(|V| + |A|)$.

------------------Solución---------------

Todo grafo conexo contiene un árbol (libre) de recubrimiento, es decir, un subgrafo acíclico conexo que alcanza a todos los vértices. Basta entonces considerar cualquiera de los vértices que están en alguno de los extremos (hojas) del árbol: si se elimina, el árbol seguirá siendo conexo. De la misma forma, si ese vértice (y sus correspondientes aristas) se elimina del grafo. este seguirá siendo conexo. Por ejemplo, en el grafo de la Figura 9.4(a). si eliminamos el vértice 4 el grafo queda inconexo, pero todos los demás se pueden eliminar.

El recorrido en profundidad de un grafo conexo (véase el Ejercicio 9.8) permite obtener un árbol de recubrimiento del grafo. El último vértice recorrido es una hoja de dicho árbol y será uno de los vértices buscados. Con una representación del grafo mediante listas de adyacencia, se puede realizar el recorrido en profundidad con el coste solicitado en el enunciado.

En realidad, lo que se pide en este ejercicio es demostrar que en un grafo

conexo no todos los vértices son *puntos de articulación.* En el Ejercicio 9.12 se define esta clase de vértices y se da un algoritmo para encontrarlos.

El alcalde de Tecno City, siempre atento a las nuevas tecnologías, ha instalado una red de comunicaciones entre los locales más emblemáticos de la ciudad, como son la alcaldía, la oficina de correos, la comisaría o el bar de la plaza mayor. La red consiste en cuerdas que entran y salen por las ventanas de los locales escogidos; a través de estas cuerdas, mediante un sofisticado sistema de pinzas y poleas, el encargado correspondiente puede transmitir mensajes codificados en celulosa.

Aunque la red no es completa (no hay cuerdas entre cada par de locales), es conexa (se puede transmitir desde cualquier local a todos los demás, posiblemente a través de locales intermedios). Lo que le preocupa al alcalde es si la red es suficientemente robusta en los siguientes aspectos:

(a) el resto de la red puede seguir funcionando aunque el encargado de algún local caiga enfermo y se quede en casa, y

(b) todos los locales seguirán estando comunicados aunque algún gamberrete corte una de las cuerdas.

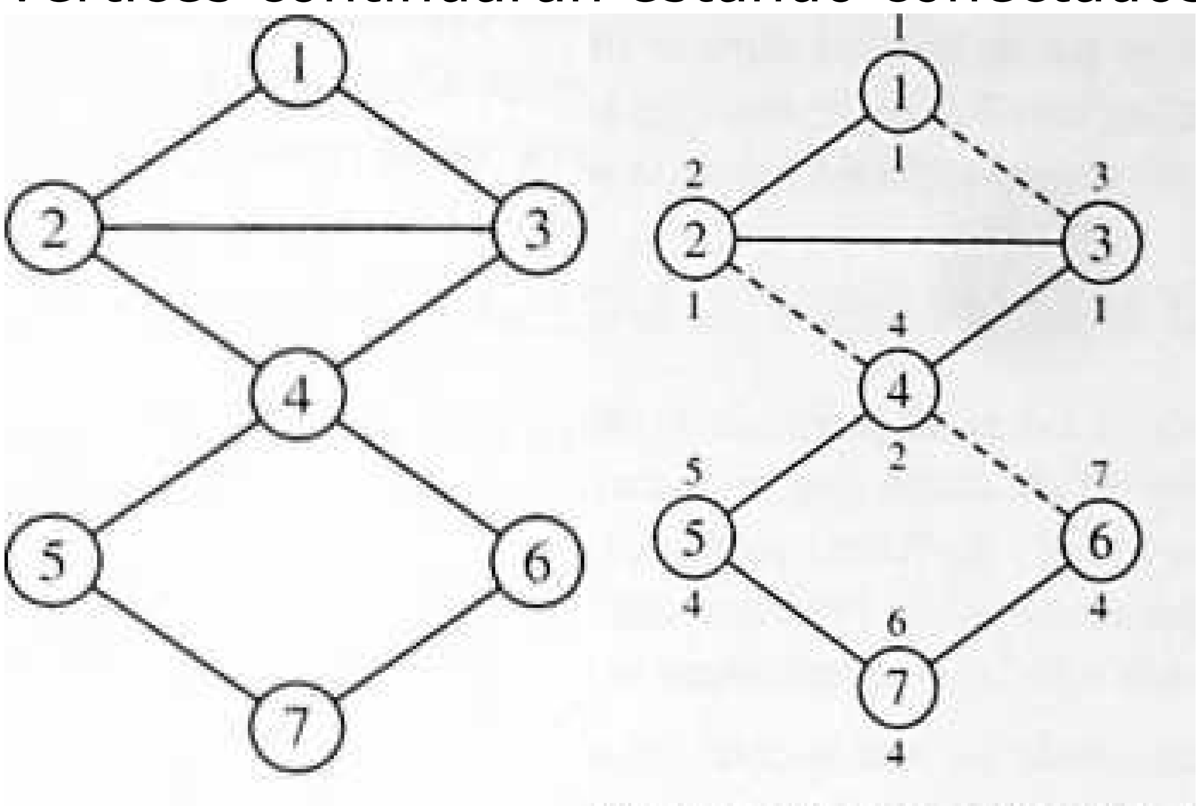
-------------------Solución**----------------**

La red de comunicaciones de Tecno City corresponde a un grafo no dirigido conexo y sin auto-aristas, donde los vértices son los locales y las aristas son las cuerdas.

Apartado (a)---------

Se dice que un vértice de un grafo

conexo es un *punto de articulación* si el subgrafo que se obtiene al eliminar dicho vértice (y sus aristas) ya no es conexo. Por ejemplo, en el grafo de la Figura 9.4(a) el vértice 4 es un punto de articulación. Así que en este apartado se desea averiguar si el grafo es *no articulado,* es decir, que no posee puntos de articulación. Cuando un grafo conexo es no articulado se dice también que es *biconexo.* porque entre cada par de vértices existen al menos dos caminos de vértices disjuntos. De esta forma, si por alguna razón uno de los caminos entre dos vértices no se puede utilizar, los vértices continuarán estando conectados.



(a) Grafo no dirigido

(b) Recorrido en profundidad empezando por el 1

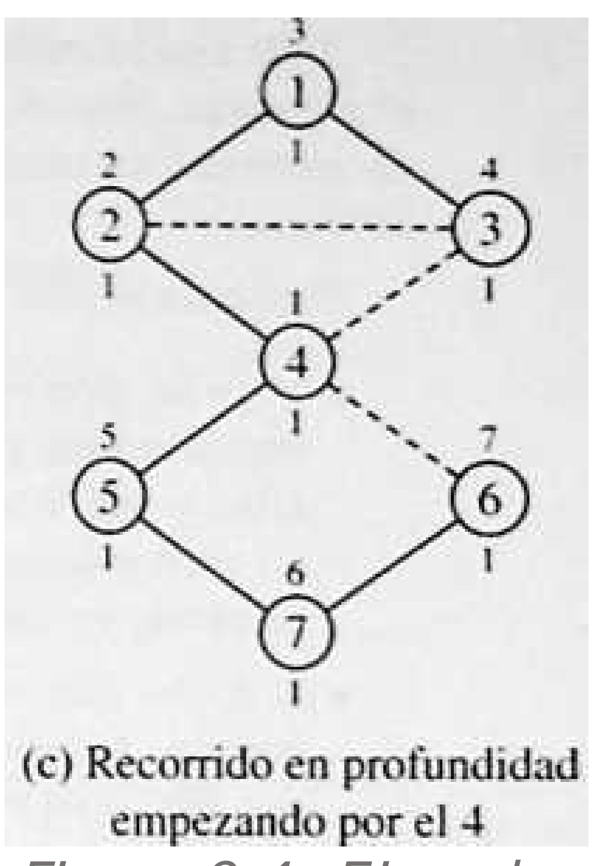


Figura 9.4: Ejemplo de punto de articulación.

Para encontrar los puntos de articulación de un grafo conexo hay que considerar un recorrido en profundidad del grafo (véase el Ejercicio 9.8). Por ejemplo, en las Figuras 9.4(b) y (c) se muestran dos recorridos en profundidad del grafo de la Figura 9.4(a): el primero comenzando por el vértice 1 y el segundo comenzando por el vértice 4. Las líneas discontinuas corresponden a las aristas que no forman parte de cada recorrido y los números que aparecen *encima* de cada vértice corresponden al orden del recorrido correspondiente.

Un recorrido en profundidad puede interpretarse como un árbol general que recubre el grafo. donde la raíz es el vértice desde el que se inicia el recorrido. De esta forma, se tiene que las aristas descartadas (líneas discontinuas en el dibujo) nunca pueden atravesar de una rama a otra del árbol, puesto que todo par de vértices conectados en el grafo de

partida deben pertenecer a la misma rama del árbol en el recorrido. Se deduce que si la raíz tiene más de un hijo (como ocurre en el recorrido de la Figura 9.4(c)). el vértice inicial es un punto de articulación, ya que si es eliminado no se podrá pasar de una rama a la otra.

En cuanto al resto de los vértices del grafo, debemos analizar si al eliminar algún vértice la parte inferior de la rama a la que pertenece dicho vértice se queda aislada o. por el contrario, las líneas discontinuas permiten "reparar" la situación. Para realizar este análisis, se asocia a cada vértice r un valor mcú-a/ro[t’) que indica el vértice "más alto" del árbol que se puede alcanzar desde *v* descendiendo primero por la rama del árbol y ascendiendo después como máximo una vez por una línea discontinua. El vértice más alto alcanzable de esta forma tiene que ser un ascendiente de v (los descendientes no se consideran porque ti ya es más alto) ya que hemos explicado antes que las líneas discontinuas no cruzan las ramas. En las Figuras 9.4(b) y (c) los números que aparecen *debajo* de cada vértice corresponden a los valores de *más-alto.* para los cuales se han utilizado los valores asignados a cada vértice durante el correspondiente recorrido.

Consideremos cualquier vértice u excepto la raíz. Si r no tiene hijos en el árbol (es una hoja), entonces no puede ser punto de articulación del grafo.

puesto que si se elimina, el resto de los vértices seguirán estando conectados (véase el Ejercicio 9.11). Sea entonces ir un hijo de r; si *más-alto[*ir | está por encima de u, eso significa que podemos partir de *u*> y llegar a una zona del árbol por encinta de r sin tener que atravesar la arista (u, ir). En consecuencia, si se elimina r. el subárbol cuya raíz es ir no queda desconectado del resto del árbol. En cambio, si *más-alto[u*] coincide con r o está por debajo de r. eso significa que la única forma de ascender por el árbol desde ir y por encima de r es atravesando la arista (v. u>). Así que. si ti es eliminado, iw y todo el subárbol correspondiente quedarían aislados.

De todo lo anterior se deduce que u es un punto de articulación si y solo si:

- t> es la raíz y tiene más de un hijo, o
- *v* no es la raíz, y tiene algún hijo ir tal que i»(ís-<i/ro[ii'] coincide con r o está por debajo de r.

Toda esta información se puede obtener durante el recorrido en profundidad del grafo. Modificamos el algoritmo dado en el Ejercicio 9.8 de forma que al visitar un vértice, además de asignarle su orden de recorrido, se calcule su correspondiente valor de *más-alto y* se determine si se trata de un punto de articulación.

Para cada vértice *w* adyacente al vértice v que se está visitando, se tienen dos posibilidades:

- u: ya ha sido visitado y no es el padre de v (si *w* fuera el padre de *v* entonces la arista (*v. w}* formaría parte del recorrido como (tu. v)): eso quiere decir que *w* es un ascendiente de *v* en el árbol y que la arista *(v. w)* no forma parte del árbol de recorrido. Así que más-o/zo[v] tiene que ser menor o igual que *recorrido\u>]* (nótese que si *v y ui* aparecen en la misma rama del árbol del recorrido y v está "más alto" que *u>.* entonces se tiene que *recorrido[v] < recorrido[w]).*
- *w* no ha sido visitado todavía; eso quiere decir que *w* va a ser un hijo de v en el árbol de recorrido. Así que *más-alto[v]* tiene que ser menor o igual que Además, en caso de que *más-alto[w] > recorrido[v],* ya sabremos que *v* es un punto de articulación. El parámetro *articulación* nos permite abandonare!

recorrido en cuanto se detecte uno de estos puntos.

Todo esto no es aplicable a la raíz, que es el vértice inicial del recorrido, y debe ser tratada de forma especial, controlando si genera más de un hijo.

proc DFS-articulación(e *G : grafojn],* e *padre, v : \..n.k* : 1..*n, recorrido[* 1. u] **de** O..zi.

zrzíís-o/zo[l.jr] de 1..*n,*
articulación : bool)

var / : *lista[\..n]*

k := k **+ I ;** *recorrído[v\ k* **; z?iás-a/zo[v] :=** *k*

l := adyacentes(v, G)

mientras -'es-lista-vacía? (/) **hacer**

w := izquierdof/) ; elim-ízq(Z)

si *-•articulación* **entonces**

si r<?cornWo[ui] = 0 entonces

DFS-articulación(G. v. *w. k, recorrido, más-alto, articulación)*

más-alto[v] := mín(/z/<ís-rz/Zo|u]. *más-alto[w])*

articulación := nzds-a/zo[w] > *recorrido[v]*

si no

si *w* ± *padre* **entonces**

(w no es el padre de *v*

} mós-a/to[v] := mín(zzzós-fl/Zo[v], *recorrido[w])* **fsi**

fsi

fsi

fniientras

fproc

El algoritmo para determinar si el grafo (conexo) es biconexo se completa con la inicialización de los parámetros para realizar la llamada inicial del recorrido y con la generación de los hijos del vértice inicial. En realidad solamente se genera uno. porque sí se intenta generar otro hijo se determina que el grafo es articulado. Esto se consigue usando la variable booleana *primer-hijo* en el algoritmo que viene a continuación.

La determinación de los puntos de articulación es independiente del vértice inicial escogido para el recorrido. En el algoritmo siguiente comenzamos el recorrido por el vértice 1.

fun biconexo?(G : *grafo[ni)* dev *b : bool*

var *recorrido[\..n]* **de O..n,** *más-alto[* **1 ..zz] de 1** *..n, l : Hsta[\..n]*

recorrido[2..n] := [0]

recorrido[**1] := 1 ;** *más-alto[* **1] := 1**

k := 1 ; *articulación :*= falso ; *primer-hijo* := cierto

/ := adyacentes! 1. G)

mientras -’es-lista-vacía?(/) **hacer**

v := izquierdo(Z) ; elim-izq(Z)

si -•*articulación recorrido[v] = 0* **entonces**

si *priiner-hijo* **entonces**

DFS-articulación(G. I, i>.

k. recorrido, más-alto,

articulación) priiner-hijo := falso **si no** *articulación* := cierto **fsi**
fsi
fmientras
b := —•*articulación* **ffun**

El coste de biconexo? es del mismo orden que el del recorrido en profundidad (0(n + *ni).* siendo *m* el número de aristas, para listas de adyacentes y 0(rr) para la matriz de adyacencia), ya que el trabajo adicional para cada vértice es de coste constante.

Apartado (b)-----------------

En este apartado se desea averiguar si el grafo es *bicoherente* (o *arísta-bicone.xo).* Por ejemplo, el grafo de la Figura 9.4(a) es bicoherente. pero el de la Figura 9.3(a) no lo es.

Si el grafo es biconexo se asegura la conexión entre cada par de vértices aunque se elimine alguna arista. Pero aunque esta condición es suficiente, no es necesaria para ser bicoherente.

Por una parte se necesita que cada vértice tenga al menos dos aristas (que no sean auto-aristas; para simplificar la presentación, vamos a suponer que en el grafo no hay este tipo de aristas), porque en caso contrario quedaría aislado si falla su única conexión con el resto del grafo. Por otro lado, es preciso que todo punto de articulación esté unido mediante al menos dos aristas

con cada componente del subgrafo restante.

Por ejemplo, el recorrido de la Figura 9.4(b) muestra que el vértice 4 es un punto de articulación, pero que además de la conexión con el vértice 3 para la componente superior y con el vértice 5 para la componente inferior, ofrece conexiones alternativas a través de los vértices 2 y 6. respectivamente.

Consideremos de nuevo el árbol de recubrimiento generado por un recorrido en profundidad. Se observa que cada punto de articulación (que no sea la raíz) "separa" el árbol en una zona superior (los ascendientes y las otras ramas del árbol) y en otra inferior (los descendientes). De un punto de articulación *v* parten dos aristas en el árbol, una hacia la zona superior y otra hacia la zona inferior. Para que el grafo sea bicoherente. r debe ofrecer alternativas (líneas discontinuas) para pasar a cada una de estas zonas. Esto corresponde a encontrar vértices adyacentes a r que no se visitan desde r. En concreto, se necesita encontrar dos: uno que corresponda a la zona superior (su orden de recorrido será anterior al de r) y otro que corresponda a la zona inferior (su orden de recorrido será posterior al de r). Dicha comprobación se puede realizar durante el recorrido en profundidad.

Resumiendo, durante el recorrido (en

profundidad) se debe controlar que cada vértice tenga al menos dos adyacentes y que todo punto de articulación tenga doble conexión con cada zona.

proc DFS-bicoherente(e *G : grafo[n],* e *padre,* i» : l..n. *k* : l.ai. re<«rrí</o[l..n] de O..n.

niás-alto[l ./i] de *l..n.*

bicoherente : bool)

var *i lista[\..n]*

k := *k* + 1 : r<'corr«/o[i>] := *k ;* »κÍS-<Í/W[I>] := *k*

a := 0 (« contabiliza las aristas que parten de v)

articulación := falso

superior := falso ; *inferior* := falso { indican si hay conexión alternativa |

/ := adyacentes(i'. G)

mientras -■es-lista-vacía?(Z) **hacer**
tu := izquierdo(Z) ; elim-izq(Z)
a := a + 1
si *bicoherente* **entonces**
si *recorridofw]* = 0 **entonces**
DFS-bicoherente(G. o, *w. k, recorrido, más-alto, bicoherente)*
más-alto[v] := mín(inás-alto[v], inás-alto[w])
articulación := articulación V más-alto[w] > recorrido[v]
si no
si *w padre* **entonces**
superior := recornc/o[io] < *recorrido[v]*
inferior := recorrido[w] > recorrido[v]
más-alto[v] := mín(niás-alto[v], recorrido[w])
fsi
fsi
fsi
fmientras
bicoherente := bicoherente A « > 2 A *(—'articulación* v *(superior* A *inferior))*
fproc

En general, de la raíz (como el grafo es conexo se puede elegir cualquiera, por ejemplo el vértice 1) cuelgan varios subárboles. Para que el grafo sea bicoherente es necesario que el vértice raíz ofrezca más de una conexión con cada subárbol. Esto se controla mediante la variable booleana *otra,* que para que el grafo sea bicoherente debe ser igual a cierto

cada vez que se pasa a un nuevo subárbol en el algoritmo que viene a continuación.

fun bicoherente?(G : *grafo[n])* **dev** *b : bool*

var recorrúZo[l..zi] **de** O..n, m«s-«/ro[l.jt] de I *,.n. I : listaf l..n]*

recorrido[2..n] := [0]

recorrido))) 1 : *inás-alto[* 1] := 1

k := 1 ; *b* := cierto ; *otra* := cierto

l := adyacentes(l, G)

mientras -’es-lista-vacia?(Z) **hacer**

v := izquierdo(Z) ; elim-izq(/)

si *b* **entonces**

si *recorrido[v]* = 0 **entonces** (nuevo subárbol)

b := otra ; otra := falso

DFS-bicoherente(G. I. v. *k, recorrido, más-alto, b)*

si no *otra* := cierto (otra conexión en el mismo subárbol)

fsi

fsi

fmientras

b := b A *otra* (comprobar el último subárbol) **ffun**

Como en el apartado anterior para biconexo?. el coste de bicoherente? es del mismo orden que el del recorrido en profundidad *(<d(n* + »;), siendo *ni* el número de aristas, para listas de adyacentes y 0(/r) para la matriz de adyacencia).

Escribir un algoritmo que determine las componentes fuertemente conexas (subgrafos fuertemente conexos con el

mayor número de nodos, véase la Sección 9.1) de un grafo dirigido.

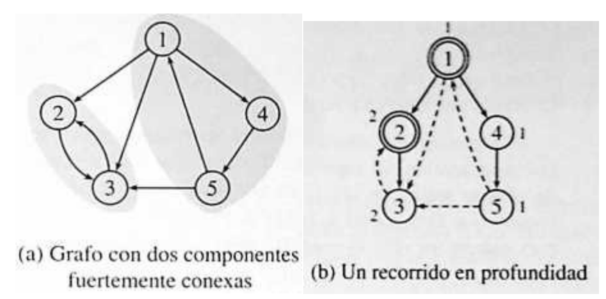


Figura 9.5: Ejemplo de componentes fuertemente conexas.

------------------Solución------------------

Los vértices de un grato dirigido se reparten en componentes fuertemente conexas de forma única, y cada vértice pertenece exactamente a una componente. Como caso extremo tenemos que cada componente fuertemente conexa contenga un único vértice, como ocurre en el grafo de la Figura 9.2. En la Figura 9.5(a) se muestra un grafo con dos componentes fuertemente conexas, una formada por los vértices 2 y 3. y otra por los vértices 1.4 y 5.

Consideramos una vez más un recorrido en profundidad del grafo *completo* (véase el Ejercicio 9.8). Dicho recorrido determina un *bosque de arborescencias* (definidas en el Ejercicio 9.10). y todos los vértices de una misma componente fuertemente conexa pertenecen a la misma arborescencia. Entonces, para cada componente fuertemente conexa denominamos *raíz de la componente* al vértice que está más alto en la arborescencia. De esta forma, la raíz, es el primer vértice de la componente que se visita durante el recorrido en profundidad. Si somos capaces de identificar estas raíces, podremos

determinar la partición en componentes.

En la Figura 9.5(b) se muestra un recorrido en profundidad del grafo de la Figura 9.5(a) desde el vértice I. En este caso el orden de recorrido coincide con la numeración de los vértices. Las aristas que no intervienen en el recorrido aparecen con flechas discontinuas. Hemos marcado con un doble círculo las raíces de las dos componentes, que son los vértices 2 y 1.

Vamos a ver que estas raíces son similares a los puntos de articulación (véase el Ejercicio 9.12). Para que un vértice *v* sea la raíz de una componente, se tiene que cumplir que no exista ninguna arista que vaya desde un descendiente de v hasta un ascendiente de v: porque si así fuera, dicha arista completaría un ciclo que conectaría u con el vértice ascendiente, el cual pertenecería entonces a la misma componente que *v. y* este último ya no podría ser la raíz de la componente.

Podemos determinar la existencia de este tipo de aristas mediante el uso de valores *más-alto,* como en la detección de los puntos de articulación. Aunque ahora hay que tener en cuenta que en los gratos dirigidos sí pueden aparecer aristas que "crucen” de una rama a otra. Estas aristas que cruzan entre ramas siempre van de "derecha a izquierda", es decir, apuntan hacia vértices ya visitados.

Supongamos que estamos buscando la raíz de la primera componente, y que

para un hijo ir de r se tiene una arista (tu. *u*) que cruza hacia el vértice *u* visitado anteriormente a r en otra rama. Entonces la raíz de la componente no puede ser *v,* sino que tiene que ser un ascendiente común de v y *u.* El hecho de que la componente que contiene a *u* no haya sido encontrada aún significa que existe alguna torma de subir desde *ti.* Así pues, las aristas que cruzan a vértices ya visitados tienen el mismo efecto que las ascendentes y no es necesario distinguirlas.

Se define *más-alto* de un vértice como el menor (los vértices más altos en la arborescencia tienen los menores valores de recorrido) entre los valores *más-alto* de sus hijos y de entre sus aristas ascendentes o que cruzan (en la Figura 9.5(b) junto a cada vértice aparece su valor *más-alto}.* Un vértice será la primera raíz si es el primer vértice para el cual su valor *más-alto* no está más alto que él mismo.

Una vez que se ha determinado la primera raíz, la componente fuertemente conexa consiste en todos sus descendientes en la arborescencia. Para encontrar la siguiente componente debemos "eliminar" la primera componente del grafo. Para ello basta marcar los vértices de la componente encontrada e ignorar las aristas que apunten a vértices marcados.

El algoritmo que damos a continuación para obtener las componentes fuertemente conexas utiliza una modificación del algoritmo de recorrido en profundidad que además de llevar cuenta del número de vértices recorridos (A), también lleva el número actual de componentes encontradas (c). El vector *componente* lleva cuenta de la componente a la que pertenece cada vértice. Asimismo se utiliza una pila para guardar los vértices pendientes de ser asignados a una componente, hasta que la raíz sea encontrada.

proc DFS-fuertemente-conexas(e *G : grafofn]*, **e** *v :* l..n, *k, c : O..n, P : pila[* I *,.n], recorrido[\..n],componente[l..n]* **de** *O..n, más-alto* 1 ,.n] **de** l.ji)
var Z : *lista[l..n]*
k k + 1 ; recom'rZo[v] := *k ;* ni«5-«/ro[v] := *k*
apilarfu, *P)*
l := adyacentesfv, G)
mientras -'es-lista-vacia?(Z) **hacer**
w := izquierdo(Z) ; elim-izq(Z)

si *recorrido[w]* = **0 entonces**
DFS-fuertemente-conexas(G. *w, k, c. P, recorrido, componente, más-alto)*
más-alto[v] := mín(mó.v-n/zo[t>].
si no
si *recorrido[w] < recorrido[v]* A
coinponente[w] **= 0 entonces** m¿Á'-a/ío[t)] := mín(,nds-oZzo[v], *recorrido[w])*
fsi
fsi
fniientras
si *más-alto[v] = recorrido[v]* **entonces**
{ *v* es raíz de una componente)
c := c + 1 ; *componente[v]* := c
w := cima(P) ; desapilar(P)
(marcar los vértices de la nueva componente |
mientras *w* / v **hacer** (se desapila hasta *v,* inclusive)
componente[w] := c
w := cima(P) ; desapilar(P)
fmientras
fsi
fproc
fun fuertemente-conexas(G : *grafo[n])*
dev *componente^ ,.n)* **de** O..n
var *P :pila[i..n], recorrido[l..n]* **de** *O..n, más-alto[l..n]* **de** l..n
P := pila-vaciaO
recorrido[l..n] := [0]; *componente[l..n]* := [0]
k := 0 ; *c* := 0
para *i* = 1 **hasta** *n* **hacer**
si *recorrido[v]* = 0 **entonces**
DFS-fuertemente-conexas(G, *v, k, c,*

P. recorrido, componente, más-alto)
fsi
fpara
ffun

El coste de fuertemente-conexas es del mismo orden que el del recorrido en profundidad, y resulta estar en 0(n + *m)* para listas de adyacentes y en $0(zi^2)$ para la matriz de adyacencia.

Diseñar un algoritmo que, dado un grafo G y un vértice inicial v, **recorra todos los vértices en *G*** accesibles desde *v,* de forma que antes **de** recorrer **un vértice a distancia *d* de r. se hayan recorrido** todos los vértices a menor distancia.

--------------------Solución------------------

Lo que se está pidiendo es un recomido en *anchura* (en inglés *Breadth First Search.* abreviado BFS). que es equivalente al recorrido *por niveles* en los árboles (véase el Ejercicio 6.12). Por ejemplo, para el grafo de la Figura 9.2, un posible recorrido en anchura desde el vértice 1 visitaría los vértices en el orden siguiente: 1,2. 3.4.5, 6,7, 8; el resto de los vértices (9, 10. 11, 12) no son visitados porque no son accesibles desde el primer vértice.

Durante el proceso de recorrido, cada vértice del grafo estará en uno de los siguientes estados: **no visitado:** todavía no ha sido alcanzado en el recorrido: **visitado:** el vértice ya ha sido recorrido, pero sus adyacentes no han sido todavía explorados: **o explorado:** además de haber sido visitado, también se han recorrido ya todos sus vértices adyacentes.

Así pues, por un lado, marcaremos aquellos vértices ya visitados, y por otro, tendremos una estructura donde almacenar los vértices visitados y pendientes de ser explorados sus

adyacentes. Para que la exploración posterior se haga en el orden adecuado, utilizaremos una cola (véase el Capítulo 4). Necesitamos una estructura FIFO porque hay que visitar antes los vértices más cercanos que son los que entraron antes en la cola. De esta forma, el recorrido comienza visitando el vértice inicial *v,* que se marca como *visitado* y se almacena en la cola; en las etapas siguientes, se extraerá siempre el primero de la cola y se explorarán todos sus vértices adyacentes: aquellos que no hayan sido visitados todavía, serán visitados y marcados como tales e introducidos en la cola.

Como en el recorrido en profundidad (véase el Ejercicio 9.S). se obtiene un vector con el orden de recorrido de cada vértice. Este vector sirve además para distinguir los vértices ya visitados, los cuales tendrán un orden mayor que 0.

Si suponemos que los vértices del grafo están fijados y representados con naturales en el conjunto {1. . .. íI), el algoritmo que implementa estas ideas es el siguiente:

proc BFS(e *G : grafo[n],* **e** *v* : l..n, *recorrido[* 1,,/tJ **de** 0..n) **var** *C : cola[l..n], l : lista[l..n] k* := 1 . recorrírío[l..n] := [0] **C := cola-vacia()** *recorridolv] :=* 1 ; visitar(v) ; pedir-vez(C. v) **mientras -■es-cola-vacía?(C) hacer** *u* := primero(C) ; avanzar(C) *I :=* adyacentes(i(, G) **mientras** -^es-lista-vacía’t/) **hacer** *w* := izquierdo(() ; elim-

izq(/) **si** *recorrido[w]* = 0 **entonces**
k := *k* + 1 ; recomWo[u>l := *k* ;
visitar(u') pedir-vez(C, *w)* **fsi fmientras**
fmientras
fproc

Nótese la similitud de este algoritmo con la versión iterativa dada para el recorrido en profundidad. Allí se utiliza una pila para almacenar los vértices que están siendo explorados, mientras que aquí se utiliza una cola. La diferencia en el comportamiento de estas estructuras (L1FO para pilas y FIFO para colas) produce la diferencia en el recorrido. En el recorrido en profundidad se apila cada vértice junto con la información de los vértices adyacentes que quedan por explorar. Esto no es necesario en el recorrido en anchura porque todos los adyacentes a un vértice se recorren en sucesión, mientras que al recorrer en profundidad se intercalan los vértices accesibles desde cada adyacente.

El algoritmo es correcto tanto para grafos dirigidos como no dirigidos (y para grafos valorados).

Lema:

Para *G* = (V. *A*) y v G V se cumple que BFS(G. u) visita todos y solamente los vértices en G accesibles desde v. Además, la visita se hace en el orden requerido; es decir, para todo *w. w'* e *V* si A(v. *w)* < A(v, ui'), la ejecución de BFS(G, *v)* visita *w* antes que u>'.

Demostración:

Se observa que en el algoritmo todo

vértice que se visita se incluye en la cola. Por otra parte, solamente se incluyen en la cola el vértice inicial u y vértices adyacentes a aquellos que, a su vez, han estado en la cola: es decir, que todos los vértices que se incluyen en la cola son accesibles desde *v.* Por lo que se deduce que todos los vértices que se visitan son accesibles desde v.
A continuación, demostraremos que todos los vértices accesibles desde *v* se visitan y son introducidos una vez en la cola, y que si *u* se introduce en la cola (y se visita) más tarde que *w,* entonces A(v, ui) < A(v. w). LO haremos por inducción sobre A(v, *w).*
En el caso básico, el único vértice a distancia 0 de *v es* el propio v. que es el primero que se visita y se introduce en la cola.
Para el paso de inducción, sea *w* € V. accesible desde *v.* tal que A(u. u>) = <7+ 1; eso quiere decir que existen un vértice *w'* 6 *V* con A(u. *w') = d* y una arista (*w'. w*) e A. Como *w'* también es accesible desde v. por hipótesis de inducción, habrá sido visitado antes e introducido en la cola. En algún momento *w'* será extraído de la cola y *w,* corno vértice adyacente a u . será visitado (si no lo ha sido antes) e introducido en la cola. Además, cualquier vértice *u* que se introduzca en la cola posteriormente a *w.* será adyacente a algún vértice *tí* introducido en la cola posteriormente a *w'.* y se

tiene que
A(u, n) = A(v, «') + 1 >Z|J A(V, UI') + 1 =
A(r, u>).
s/

Este algoritmo puede utilizarse para recorrer *todos* los vértices de un grafo de la forma siguiente: se parte de un vértice cualquiera y se aplica el procedimiento BFS; al finalizar se escoge otro vértice cualquiera que todavía no haya sido visitado. Y así sucesivamente hasta que todos los vértices hayan sido visitados.

Vamos ahora a analizar el coste del algoritmo, comenzando por su coste en tiempo. El coste de la inicialización del vector *recorrido* es lineal con respecto a n, el número de vértices en el grafo. Por otra parte, si cada vértice se añade a (y se extrae de) la cola una vez como máximo, en el caso peor, en que todos los vértices sean accesibles desde el inicial, y considerando que las operaciones básicas sobre la cola tienen un coste constante, el coste correspondiente a todo este trabajo está también en 0(«).

En cuanto al recorrido de los vértices adyacentes, su coste depende, como es usual, de la representación escogida para el grafo. Si se utiliza la matriz de adyacencia, el coste total estará en ya que para cada vértice se recorrerá por completo la fila correspondiente. En el caso de utilizar listas de adyacentes, el coste estará en *&(n + ni)* (siendo *m* el número de aristas), puesto que se

recorrerán las *n* listas, pero el tamaño total de todas las listas es *m.*
Con respecto al coste en espacio, hay que considerar el espacio ocupado por ia cola de vértices. Como cada vértice del grafo entra como mucho una vez en la cola, esta contendrá un máximo de *n* vértices, y en el caso peor, en que todos los demás vértices sean adyacentes al de partida, contendrá efectivamente *n* — 1 vértices. Por tanto, el coste en espacio adicional estará en 0(n).

Curro por fin ha vuelto de sus largas vacaciones por el Caribe y se ha encontrado sobre su mesa de trabajo un enorme montón de tareas pendientes. Curro sabe que tendrá que realizar todas las tareas él solo, ya que sus envidiosos compañeros le odian por sus maravillosos viajes. Al ir a planificar su trabajo, Curro se ha dado cuenta de que algunas tareas dependen de otras, de forma que no puede empezar con una tarea hasta que no haya completado todas las tareas de las que depende. Curro ha ido en seguida a quejarse a su jefe de que en esas circunstancias no va a poder sacar adelante el trabajo. Pero su jefe le ha asegurado que existe al menos una ordenación posible de todas las tareas, y que el primer objetivo de Curro es determinar alguna de estas ordenaciones para después ponerse a realizar las tareas en el orden calculado.

--------------------Solución------------------

La estructura adecuada para la representación de los datos es un grafo dirigido, donde los vértices son las tarcas a realizar por Curro, y estableciéndose una arista desde la tarea ,r hasta la tarea y cuando para comenzar y es necesario haber terminado antes ,r. Puesto que el jefe asegura que todas las tareas pueden planificarse, el grafo ha de ser acíclico. Lo que se pide entonces es una *ordenación topológica* de los vértices del grafo. es decir, si el vértice t> aparece en la ordenación en la posición *i.* entonces

todos los vértices accesibles desde *v* ocuparán posiciones posteriores a *i.* Para un grafo *G* = (V'. A) se pide encontrar una ordenación *O : V* —► 1N tal que para toda arista (u. *w*) e A se cumpla Oír) < O(u-). Por ejemplo, el grafo de la Figura 9.2 es acíclico, y una posible ordenación topológica es la secuencia 1.2. 4. 9. 10. 11.3.5, 12,6,8,7.

La estrategia para encontrar la ordenación consiste en ir considerando de forma reiterada vértices con grado de entrada 0, eliminándolos del grafo junto con todas las aristas que salen de dicho vértice. Puesto que el grafo es acíclico, en cada momento debe haber algún vértice con grado de entrada 0, pues de lo contrario, partiendo de cualquier vértice y recorriendo aristas en sentido inverso (del destino hacia el origen), generaríamos un camino infinito, lo cual implicaría la existencia de un ciclo en el grató. Claramente, al eliminar vértices o aristas en un grafo acíclico se mantiene dicha propiedad.

Para lograr una implementación eficiente de la estrategia propuesta, conviene guardar la información sobre el grado de entrada de cada vértice y utilizar una cola donde se van añadiendo los vértices con grado de entrada 0 que se van generando, para tratarlos posteriormente. El algoritmo resultante es:

(G es acíclico)

fun orden-topológico(G . *grafo[n])* **dev**

orJen[l..zi] de L.zz
var *grado-de-entrada[* **1..zt] de** *nat, I : lista[l..n], C : cola[\..n]*
grado-de-entrada := obtener-grados-de-entrada(G)
C := cola-vacía()
para z = I hasta *n* **hacer**
si *grado-de-entrada[i]* — 0 **entonces** pedír-vez(C. ¿) **fsi**
fpara
posición := 0
mientras -^es-cola-vacia?(C) **hacer**
v := primero(C) ; avanzar(C)
posición := posición + 1
orden[v\ := posición
I := adyacentes(v. G)
mientras -•es-lista-vacía?!/) hacer
w := izquierdo(í) : elim-izq(/)
grado-de-entrada[w] := grado-de-entrada[w] — 1
si *grado-de-entrada[w]* = 0 **entonces** pedir-vez(C. **ir) fsi**
finientras
(mientras
ffun

Como ya se comentó en el Ejercicio 9.10. la manera de obtener el grado de entrada de los vértices depende de la representación elegida para el grafo: con una matriz de adyacencia el coste está en 0(n^2), mientras que con listas de adyacentes el coste está en ®(n + ni), siendo *ni* el número de aristas.

El número de iteraciones del bucle principal del algoritmo orden-topológico coincide con el número de vértices (hay que dar un número de orden a cada vértice), pero el número total de actualizaciones del grado de entrada de los vértices coincide con el número de aristas. En cuanto a la obtención de los vértices adyacentes, hacemos la mismas consideraciones que en el Ejercicio 9.S: el coste total estará en 0(n^2) si se utiliza la matriz de adyacencia, y en 0(n + *ni)* si se utilizan listas de adyacentes. El resto del trabajo es de coste constante, si suponemos que las operaciones básicas sobre la cola tienen un coste constante.

Resumiendo, el coste total en tiempo del algoritmo orden-topológico está en *(~)(ir)* si se utiliza la matriz de adyacencia, pero está en *@(n + ni)* si se utilizan listas de adyacentes.

El coste en espacio adicional (el tamaño máximo de la cola) está en 0(n). porque en el caso peor todos los vértices tendrán grado de entrada 0.

Dada una relación de equivalencia ~ sobre un conjunto C. la clase de equivalencia de un elemento *a* e *C* es el

subconjunto de *C* que contiene a todos los elementos relacionados con *a.* Las clases de equivalencia en *C* forman una *partición* del conjunto, es decir, todo miembro de *C* aparece exactamente en una clase de equivalencia. Para decidir si *a ~~ b.* solo necesitamos comprobar si *a* y *b* están en la misma clase de equivalencia.

Especificar el tipo de las particiones para relaciones de equivalencia dinámicas sobre conjuntos finitos incluyendo las siguientes operaciones:

- crear una partición unitaria (clases de equivalencia unitarias) para un conjunto finito de elementos donde la relación es la identidad.
- fusionar en una partición las clases de equivalencia correspondientes a dos elementos *a y b* del conjunto, al añadir el par *(a, b)* a la relación, y
- buscar la clase de equivalencia a la que pertenece un elemento del conjunto.

---------------**Solución**------------------

Las operaciones que crean una relación de equivalencia identidad y que añaden pares a la relación son las constructoras del tipo partición. Podemos interpretar una partición como un grafo no dirigido, donde crear la partición unitaria corresponde a añadir todos los elementos del conjunto dado como vértices del grafo. fusionar corresponde a añadir una arista, y buscar la clase de equivalencia corresponde a determinar el conjunto de elementos accesibles desde el elemento

dado.

Las operaciones de fusionar y buscar son parciales puesto que no se puede trabajar con elementos que no pertenezcan al conjunto base sobre el que se ha realizado la partición.

Aunque en principio no se exige ningún requisito especial para los elementos sobre los que se va a definir la relación de equivalencia, como vamos a realizar búsquedas de elementos en la partición, necesitamos tener definida una operación de igualdad sobre los elementos, por lo que consideramos el parámetro *ELEM=* (véase la Sección 1.1.5).

especificación *PARTICIONES[ELEM=]*
usa *CONJUNTOS[ELEM=¡. BOOLEANOS*
tipos *partición*
operaciones
crear-partición :*conjunto* —> *partición* (constructora)
fusionar *:elemento elemento partición* —*partición*(constructora!
buscar *: elemento partición* —»*conjunto*

operaciones privadas

relación? *elemento elemento partición* —> *bool*

variables

e. f. g. e'. f : elemento

p : partición

c : conjunto

Para poder fusionar las clases de equivalencia de dos elementos, estos deben pertenecer al conjunto base de la partición.

ecuaciones

fusionarte, *f.* crear-partición(c)) = error <= ^está?(e. c) v --está?(/, c)

Las constructoras *no son libres,* porque si dos elementos ya pertenecen a la misma clase de equivalencia no es necesario fusionarlos. Para expresar la correspondiente ecuación de equivalencia utilizamos una operación privada relación? que nos indica si dos elementos están relacionados entre sí. es decir, si pertenecen a la misma clase de equivalencia. Además el orden en el que se realizan las fusiones no es relevante:

fusionarte. *p) =. p* <= relación?(e. *f. p)*

fusionarte, *f.* fusionarte', *f. p))* = fusionarte', *f.* fusionarte, *f. p))*

No se debe buscar la clase de equivalencia de un elemento que no pertenece al conjunto base de la partición. En la partición unitaria, todas las clases de equivalencia son unitarias.

buscarte, crear-partición(c)) = error <= --está?(e. c)

buscarte, crear-partición(c)) — unit(e)

<= está?(e. c)

buscarte, fusionarf/. *g. p)}* =
unión(buscar(e, *p).* unión(buscar(/. *p).* buscarlg. /?)))

<= relación?(e, *f. p)* v
relación?(e. i?, p)

buscarte, fusionan *f. g. p))* = buscarte, p)

<= ->relación?(e./. p) A -
■relación?^, *g. p)*

Por último especificamos el predicado relación?. En una partición unitaria cada elemento (del conjunto base) solo se relaciona consigo mismo. Al hacer equivalentes dos elementos e' y *f.* no solamente quedan relacionados entre sí estos dos, sino también todos aquellos elementos relacionados con e' y con *f.*

relacion?(c. *f,* crear-partición(c)) = *e* == *f* A está?(e, c)

relación?(e. /, fusionarte', *f\ p))* = relación?(e. *f. p)* v
(relación?(e. e') A relación?(/', /)) v
(relación?(e. *f)* A relación?(e’. /)) **fespecificación**

Desarrollar una implementación del TAD de las particiones especificado en el Ejercicio **9.16. supo**niendo que el conjunto base de la partición es el intervalo de números naturales **1..1V.**

(a) Utilizando un vector de naturales.

(b) Utilizando una estructura enlazada arborescente.

------------------Solución------------------

Apartado (a)

Puesto que el conjunto base de la partición es fijo y se corresponde con el rango 1..A'. se puede implementar la partición de forma muy sencilla mediante un vector de ese mismo rango, de forma que la componente í-ésima del vector indique la clase de equivalencia a la que pertenece *i.* Una posibilidad para representar las clases de equivalencia es lomar como representante de una clase el menor elemento de esa clase. Con estas ideas, el tipo representante queda de la forma siguiente:

tipos
elemento = I..N
partición[\..N] **= vector, 1 ../V] de** *elemento*
ftipos

Al crear la partición unitaria, cada elemento pertenece a su propia clase.

fun crear-particiónl () **dev** *P : partición[\..N] { Q(N))*
para *i* = **1 hasta** *N* **hacer /’[/] :=** *i*
fpara ffun

Obsérvese que, a diferencia de la operación especificada, esta función no tiene argumento alguno porque el conjunto base de la partición es fijo.

A la hora de buscar la clase de equivalencia de un elemento basta consultar la componente correspondiente del vector. En esta implementación no se devuelve la clase completa, sino solo el representante de la misma, que es el menor elemento de la clase.

fun buscar!*(e : elemento. P : parlición[* 1

..N]~) dev *c : elemento | (-)(!) |*
c := P[e]
ffun
Para fusionar las clases de equivalencia a las que pertenecen dos elementos dados hay que determinar cuál es el menor de los dos representantes para hacer la sustitución pertinente en las componentes de los elementos pertenecientes a la otra clase. Para realizar esta sustitución es necesario recorrer todo el vector.
proc fusionan**(e** *e. f : elemento. P : partlción[*I *..N]) | (~)(N))*
mín := mín(P[e], *P[f]) ; máx :=* máx(P|e], *P[f])*
para *i* **= 1 hasta** *N* **hacer**
si P[i] = *máx* **entonces** P[í] := *mín*
fsi
fpara
fproc
La utilización típica de estas estructuras es crear una partición inicial unitaria y después realizar una secuencia de operaciones de buscar y fusionar. Se realizarán un máximo de *N* — 1 operaciones de fusionar, porque después ya no queda nada para fusionar. Si el número de operaciones de búsqueda es comparable a *N.* entonces el coste total de la utilización de la estructura está en $0(A^2)$.

Apartado (b)----------------
La idea es representar cada clase de equivalencia como un árbol (general)

invertido (los hijos apuntan hacia el padre) donde la raíz es el representante de la clase: el menor elemento. La partición será un conjunto de estos árboles, es decir, un *bosque* (véase el ejemplo de la Figura 9.6).

Como la única información necesaria es el padre de cada nodo y el conjunto base de la partición está fijo, *no* se necesita memoria dinámica para representar esta estructura, sino que basta un vector de números naturales donde cada componente contiene el valor de su padre en el árbol y las raíces contienen su propio valor. De esta forma, el tipo representante definido para el apartado anterior sigue siendo válido.

Al crear la partición unitaria, cada elemento es la raíz de su propio árbol. Nótese que el algoritmo es exactamente el mismo que para la representación del apartado anterior.

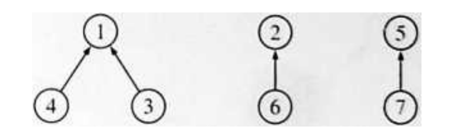


Figura 9.6: Partición ((1.3,4), (2, 6). (5, 7)) representada como un bosque.

fun crear-partición2() **dev** *P : partición! I..N]* **{ G>(7V) }**
para *i* = I **hasta** *N* **hacer** P[í| := *i*
fpara ffun

Ahora buscar el representante de la clase de equivalencia resulta ser un poco más laborioso: a partir del elemento dado, hay que "remontar" el árbol hasta llegar a la raíz.

fun buscar2(e : *elemento. P : partición! I..NJ)* **dev** *c : elemento* **(0(iV) |**
c := e
mientras /J[c] *c* **hacer** { pasara! padre)
c := P[c]
fmientras
ffun

En el caso peor, hay que recorrer un árbol degenerado de altura *N.*

En cambio, fusionar resulta muy sencillo, porque basta "colgar" de la menor raíz el árbol correspondiente a la otra clase de equivalencia. Lo que pasa es que primero hay que localizar esas raíces mediante la operación buscar:

proc fusionar2(c e, *f elemento. P : partición! l.NJ)* (0(A'))
< := buscarte. *P) : d* := buscarf/. *P)*
si *c* < *d* **entonces /’IzZ] :=** *c* **si no P[c] :=** *d* **fsi**
fproc

Ahora bien, la forma usual de utilización de estas estructuras es que se realizan las búsquedas antes de fusionar, por lo que resulta totalmente superfino volver a hacer las búsquedas en la propia implementación de fusionar: bastti hacer la llamada a fusionar con los representantes de las clases, en lugar de con dos elementos cualesquiera. De esa forma el coste de fusionar resulta constante.

Sin embargo, para una sene de operaciones consistentes en crear la partición unitaria y luego realizar *n* búsquedas entremezcladas con *N* — 1 fusiones se tiene un coste total en 0(niV) o 0(Ar–) si *n* es comparable a *N*. Es decir, que tenemos costes similares a los de la representación del primer apartado.

Si admitimos que el representante de la clase de equivalencia no es necesariamente el menor elemento de la clase, podemos acotar la altura de los árboles y mejorar el coste de la operación buscar. La idea es que al fusionar colguemos siempre el árbol con menor altura *(unión por altura).* Para que esta operación resulte eficiente llevamos un vector adicional con la altura de cada árbol.

El tipo representante queda ahora de la forma siguiente:

tipos

elemento = 1..AT

partición! 1.. *N]* = **reg**

padrel **l ../V| de** *elemento*

alrural 1. **/V] de O.Ai**
freg
ftipos

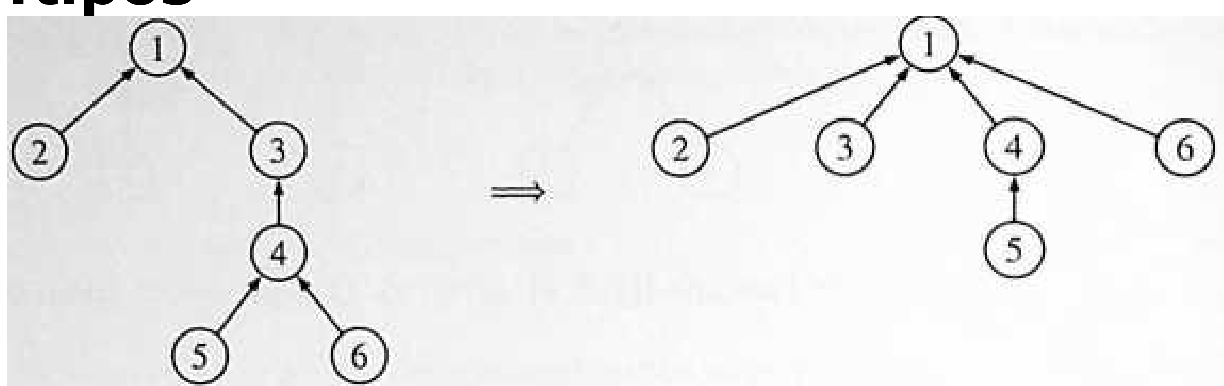


Figura 9.7: Compresión de caminos tras buscar el elemento 6.

Inicialmente todos los árboles están formados por una única hoja y tienen altura 1.

fun crear-partición3() dev *P : partición! I..N]* { 0(W) }
para *i* = 1 **hasta** *N* **hacer**
P.padre[i] := i ; P.altura[i] := 1
fpara
ffun

Al fusionar hay que modificar adecuadamente la altura en la raíz de la cual se ha colgado el otro árbol. En realidad, la altura solo se incrementa en 1 si los dos árboles coinciden en altura.

{ e y *f* son raíces }
proc fusionar3(e *e. f* ■*. elemento, P : partición!I..NJ)* (0(1))
casos
P.altura[e] > *P.altura[f] -> P.padre[f] := e*
□ *P.altura[e] = P.altura[f] —> P.padre[f] := e ; P.allum\e] := P.altura[e]* + 1
0 *P.altura[e]* < *P.altura[f] —> P.padre[e] := f*
fcasos

fproc

De esta forma se puede demostrar que la altura de los árboles no sobrepasa I + log *N* [BB97, Sección 5.9]. Para una creación inicial, *n* búsquedas *y N* — 1 fusiones se tiene ahora un coste total en 0(n log *N* + A), que cuando n es comparable a *N* equivale a *Q(N* log A).

Se puede mejorar más aún esta representación mediante lo que se denomina *compresión de caminos.* que consiste en hacer que todas las componentes recorridas durante una búsqueda terminen apuntando directamente a la raíz de su árbol, como se muestra en el ejemplo de la Figura 9.7. Ello implica hacer el recorrido dos veces, pero reduce la altura del árbol y por tanto las siguientes búsquedas serán más rápidas. Con un número suficiente de búsquedas, se tiende a que cada componente apunte directamente a la raíz, con lo que se consigue un tiempo "constante" para buscar.

Al realizar la compresión de caminos, el vector *altura* ya no contendrá la altura exacta del árbol, pero como esta técnica reduce la altura, *altura* siempre será una cota superior de la altura real.

El algoritmo ya no puede ser una función, porque debido a la compresión ahora la partición puede quedar modificada.

proc buscar3(c *e : elemento, P : partición!* 1 *,.N], c : elemento)*

c := e

```
mientras P.padre[c] / c hacer
  c := P.padre[c]
fmientras
```

d := e
mientras *d c* **hacer**
aux := P ,padre[d]
P.padre[d] c d := aux
(mientras
fproc

Para lina secuencia arbitraria de *n* llamadas a buscara y *m* < jV — 1 llamadas a fusionara se tiene un coste en *&(ca(c, N))* para *c = n + in.* donde *a(i, j)* es una función de crecimiento tan lento que, salvo para valores astronómicos de *j,* se tiene que *a(i, j)* < 4 [BB90. CLRS01]. Así que a efectos prácticos, se puede suponer que el coste en el caso peor es lineal en *c.*

Para ganar en claridad, en esta solución hemos optado por utilizar un vector para guardar la altura de cada árbol y subárbol. En realidad no es necesario este gasto extra en espacio, porque solamente se necesita saber la altura de las raíces, las cuales no tienen padre. Por eso podemos utilizar el vector donde se almacena el padre de cada elemento para guardar la altura de cada árbol en la componente de **la raíz.** Para distinguir padres de alturas, podemos almacenar las alturas con valores negativos.

Una alternativa, que no detallamos, para acotar la altura de los árboles es fusionar mediante la *unión por tamaño,* que consiste en poner como hijo el árbol de menor tamaño.

En la República de Fanfanisflán las relaciones familiares **son**

extremadamente importantes. Si uno es pariente (aunque sea lejano) de un alto cargo en algún ministerio, siempre puede conseguir algún puesto cómodo de funcionario o que sus diligencias en dicho ministerio sean **tramitadas de forma** urgente, para tener bien claro quién debe hacer favores a quién y no incurrir **en enojosos equívocos,** el primer mandatario de la República ha decretado que se elabore una partición de toda la población de Fanfanisflán en familias disjuntas. Dicha partición se ha de obtener a partir de las relaciones de parentesco directo entre los ciudadanos.

-----------------**Solución**------------------

La información sobre el parentesco entre los ciudadanos puede representarse mediante un grató, donde los vértices son los ciudadanos y se tiene una arista (n. *b)* si y solo si *a* y *b* son parientes directos. Como "ser pariente directo" es algo recíproco, el grafo es no dirigido. A partir de ese grafo de parentescos directos, lo que se desea es determinar quiénes están emparentados aunque sea de forma lejana, es decir, hay que determinar las *componentes conexas* del grafo.

En el caso de los gratos no dirigidos. la condición de ser accesible desde un vértice es una relación de equivalencia (reflexiva, simétrica y transitiva), por lo que podemos utilizar las estructuras de partición especificadas en el Ejercicio 9.16 para obtener las componentes

conexas de un grafo no dirigido en forma de una partición de los vértices. El algoritmo crea una partición inicial donde cada componente contiene un único vértice, y para cada arista del grafo se fusionan las componentes de sus extremos.

Suponemos, como viene siendo habitual, que los vértices del grafo están fijados y representados con los números naturales I.JI. Podemos utilizar entonces una de las implementaciones de las estructuras de partición dadas en el Ejercicio 9.17; en concreto utilizaremos la última, que garantiza unos costes eficientes. El algoritmo queda como sigue:

tipos
arista = **reg**
origen : vértice
destino : vértice
freg
ftipos
fun componentes-conexas (G : *grafo[n])*
dev *P : partición[l..n]* **var / :** *lista[arista], a : arista*
P := crear-partición3()
I := nd-aristas(G)
mientras -■es-lista-vacía?(/) hacer
a := izquierdo(/) ; elim-izq(Z)
c := buscar3(a.*origen, P) ; d* := buscar3(n.(/esríno, *P)*
si *c / d* **entonces** fusionar3(c. *d. P)* **fsi**
fmientras
ffun

La función nd-aristas obtiene la lista con

todas las aristas del grafo (véase el Ejercicio 9.7). En este caso no es relevante si se obtienen las dos versiones de cada arista no dirigida. Su ejecución supone un coste en 0(/r) en el caso de tener una matriz de adyacencia, y en *&(n + ni)* para las listas de adyacentes, siendo *ni* el número de aristas.

El coste de crear la partición está en 0(n). En cuanto al bucle, se realizan *ni* iteraciones, lo que supone *2in* búsquedas en la partición y *n* — 1 fusiones en el caso peor (obtener una única componente conexa). Como ya se detalló al final de la solución del Ejercicio 9.17. la representación que hemos utilizado aquí permite que el coste de estas *c = 2m +* (n — 1) operaciones sea prácticamente lineal en *c.*

Resumiendo, se puede obtener un coste lineal en *n + in* con las listas de adyacentes y cuadrático en *n* para la matriz de adyacencia.

Capitulo 10

10. APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

Vamos a recapitular aquí las ideas desarrolladas hasta ahora, presentando varios ejercicios de aplicaciones de tipos abstractos de datos.

Para cada ejercicio se pide en primer lugar la especificación de un sistema, cuyas características varían según la aplicación Esta especificación se hace siguiendo la metodología de constructoras presentada en el Capítulo I. tal y como se ha visto en los ejercicios de ese capítulo y en las especificaciones de diversos tipos de datos clásicos o variaciones de los mismos en los otros capítulos.

A continuación se pide la implementación del mismo sistema. Las implementaciones vistas en capítulos anteriores se basaban típicamente en construcciones de tipos básicas como vectores y registros o en memoria dinámica. De vez en cuando hemos visto algún ejemplo donde un tipo de datos se implementaba utilizando tipos de datos definidos previamente, como por ejemplo la implementación de colas usando pilas (Ejercicio 4.6) o de colas medievales

usando colas ordinarias (Ejercicio 4.11). Pues bien, en los ejercicios de este capítulo la idea fundamental es elegir apropiadamente los tipos de datos definidos previamente que se van a utilizar en la implementación según las características de la aplicación que se trate: colas, listas, tablas, árboles, etc. Dos aspectos fundamentales para esta elección son la forma de tratar los datos (por ejemplo, si tienen que estar ordenados o no) y la eficiencia de las operaciones que se desea implementar.

Una vez elegida la representación del sistema en términos de los tipos abstractos apropiados, se pasa a desarrollar los algoritmos correspondientes a las operaciones del sistema en cuestión. En general, explicamos todas las operaciones pero no escribimos los algoritmos detallados de todas ellas, sino solamente de las más interesantes.

Finalmente, se proporciona en cada ejercicio información sobre el coste de las operaciones implementadas. bajo las suposiciones apropiadas de coste de las operaciones de los tipos abstractos usados en la implementación.

Algunos de los ejercicios propuestos tienen apartados en los que se consideran diferentes variaciones de la misma aplicación, de forma que se vea mejor en la solución cómo afectan las características y las diferentes operaciones a la elección de la

implementación más apropiada.

EJERCICIOS RESUELTOS

10.1. Severino del Pino, profesor de arameo de la Universidad Imponente, ha detectado problemas de aburrimiento entre su numeroso alumnado. Su colega Tadeo de la Tecla, del Departamento de Informática, ha ofrecido ayudarle diseñando un sistema informático de control de bostezos. Tadeo propone especificar un tipo abstracto de datos con las siguientes operaciones:

- crear un sistema de bostezos vacío,
- registrar un nuevo bostezo en el sistema.
- borrar del sistema todos los bostezos registrados de un alumno dado,
- consultar el número de bostezos de un alumno registrados en el sistema, y
- calcular la lista de todos los alumnos que tengan tres o más bostezos registrados, ordenada por alumnos.

(a) Especificar e implementar el TAD, teniendo en cuenta que el orden en el que se registran los bostezos no es importante. No debe haber información sobre alumnos que nunca hayan bostezado.

(b) Al profesor Severino le ha encantado el sistema diseñado, pero ahora cree que su funcionalidad está algo limitada, porque también le gustaría registrar otros tipos de faltas que sus alumnos suelen cometer: tirarle tizas, hacer volar aviones de papel, escapar por la ventana, etc. Además le gustaría poder decidir en cada momento cuál es el número de faltas que ha tenido que cometer como mínimo un alumno para aparecer en la lista que calcula la última operación. Extender la especificación e implementación del apartado anterior adecuadamente.

---------------Solución:---------------

Apartado (a)

El TAD que especifica el sistema de bostezos está parametrizado sobre el tipo de los alumnos. Como es necesario realizar búsquedas por alumno, este tipo tiene que tener una operación de igualdad. Además, como una de las operaciones devuelve una lista ordenada por alumnos, este tipo también debe tener una operación de orden. Utilizaremos las listas ordenadas especificadas en el Ejercicio 5.7.

parámetro *ALUMNOS*
usa *BOOLEANOS*
tipos *alumno*
operaciones
_ == _ : *alumno alumno —► bool*
_: *alumno alumno —> bool*
_ <_ : *alumno alumno —> bool*
_ >_ : *alumno alumno —> bool*
variables
x, y, z : *alumno*
ecuaciones
(x == y) = cierto <= *x* = y
x = y <= (x == y) = cierto
x / y = --(x == y)
x < x =cierto (reflexividad)
x < y =cierto<=x< z A z < y (transitividad)
x > y = -(x < y)
x > y = falso<= y > x (antisimetría)
fparámetro

especificación *BOSTEZOSfALUMNOS]*

usa *NATURALES, LISTAS-ORDENADAS[ALUMNOSi*
tipos *fichero*
operaciones
crear —» *fichero* { constructora)
bostezar*alumno fichero* —> *fichero* (constructora 1
borrar*alumno fichero* —> *fichero*
cantidad*alumno fichero* —» *nat*
lista-negra : *fichero* —► *lista[alumno]*
variables
.v. *y : alumno*
f : fichero

Las operaciones constructoras son crear y bostezar. Estas constructoras *no son libres,* ya que el orden en el que bostezan los alumnos no es importante, por lo que tendremos una ecuación de equivalencia que muestre este hecho. El resto de las operaciones se definen distinguiendo casos según las constructoras, y utilizando la operación de igualdad entre alumnos.

ecuaciones
bostezaría-, bostezarfv. /)) = bostezaría, bostezaría, /)) (conmutatividad]
borrarla , crear) = crear
borrarla-, bostezaría-,/)) = borrarla,/)
borrarl v. bostezarí a,/)) = bostezarfy. borrarla,/)) <= a v
cantidadla. crear) = 0
cantidadla . bostezaría ,/)) = 1 + cantidadla,/)
cantidadla . bostezar!y,/)) =

cantidadla,/) <= a £ y

Para obtener la *lista negra* (ordenada) de alumnos que han bostezado tres o más veces, utilizamos la operación insertar-ord que inserta ordenadamente un elemento en una lista ya ordenada (véase el Ejercicio 5.7). Solo los alumnos que han bostezado más de dos veces se añaden a la lista, y solo se añaden una vez. Esto se consigue eliminando del sistema de bostezos utilizado para construir el resto de la lista aquel alumno que sabemos va a estar en la lista.

lista-negra(crear) = | |
lista-negral bostezaría./)) = insertar-ordf.r. Iista-negra(borrar(.v. /))) s= cantidadla./)> 2 lista-negral bostezaría-, /)) = lista-negra(/) <= cantidadla./)< 2

(especificación

En cuanto a la implementación de este TAD, necesitamos una estructura donde esté calculado, para cada alumno, el número de bostezos que ha realizado, y solo interesa tener información de alumnos que hayan bostezado alguna vez. Como se realizan consultas por alumnos, interesa que la búsqueda de un alumno sea eficiente. Por tanto, una representación apropiada consiste en utilizar una *tabla ordenada* implementada mediante un árbol de búsqueda (véanse los Ejercicios 7.7 y

7.8) donde los alumnos se utilicen como claves y la información asociada a cada clave sea el número de bostezos, may or que 0.

tipos

fichero = tabla[alumno, nat⁺]

ftipos

Crear el sistema de bostezos vacío corresponde a crear la tabla vacía. Para registrar un nuevo bostezo basta con insertar en la tabla el alumno con valor asociado 1. Si el alumno no estaba en la tabla, será este valor el que se añada, y si el alumno ya estaba se utilizará la operación combinar, que en este caso corresponde a la suma de números naturales.

proc combinar(n i *: nat.* **c** *112* : naí)
ni *:= n* + ni
fproc
proc bostezare *a : alumno, f : fichero)*
insertar-tabla(n, 1, /)
fproc

Para borrar del sistema los bostezos de un alumno se elimina el alumno de la tabla con la operación eliminar-tabla con el alumno en cuestión.

Para consultar el número de bostezos de un alumno, primero hay que averiguar si el alumno está en la tabla. Si está, hay que consultar el valor asociado, y si no está, el número de bostezos es 0.

fun cantidadfn : *alumno, f : fichero)*
dev n : *nat*

si está-tabla?*(a. f)* entonces n := consultar-tablafn. *f)*
si no n := 0
fsi
ffun

La lista negra se construye a partir del recorrido ordenado de la tabla, obtenido utilizando la operación recorrer-ordenada (que devuelve una lista ordenada por claves) y *filtrando* los alumnos con menos de tres bostezos. Hay que tener en cuenta que la lista devuelta por recorrer-ordenada es una lista de pares *(alumno, número de bostezos).* mientras que la lista devuelta por lista-negra es solo una lista de alumnos.

tipos
par-bostezos **= reg**
alumno : alumno
núm-bostezos : nat
freg
ftipos
fun lista-negra(/ : *fichero)* dev / : *lista[alumno]*
var *Ip : lista[par-bostezos]*
Ip := recorrer-ordenada*(f) ; /* := lista-vacíaO
mientras -'es-lista-vacía?(/p) **hacer**
p := izquierdo(/p) ; elim-izq(/p)
si *p.núm-bostezos* > 3 **entonces**
añadir-der(/. *p.alumno)* **fsi**
fmientras ffun

Suponiendo que la tabla se ha implementado mediante un árbol de

búsqueda equilibrado, la operación crear tiene un coste constante. El coste de las operaciones bostezar, borrar y cantidad es logarítmico respecto al número de alumnos en la tabla. Suponiendo que las operaciones utilizadas sobre las listas tienen un coste constante, el coste de lista-negra es lineal respecto al número de alumnos en la tabla, ya que los alumnos se mantienen en orden en el árbol.

Apartado (b)----------------

Para realizar la generalización es importante saber cuántos tipos de faltas diferentes puede cometer un alumno. Si el número es constante y pequeño podernos añadir operaciones constructoras que registren cada tipo de falta, y de la misma forma, tener operaciones de consulta y eliminación para cada tipo de falta. Si el número de tipos de faltas no es constante o conocido de antemano, entonces conviene abstraer y tener un nuevo tipo *falta* cuyos valores serán los diferentes tipos de falta. De esta manera el sistema de bostezos estará parametrizado por los alumnos y por las faltas. En cuanto a las operaciones modificadoras y observadoras, tendremos para cada una dos versiones: una que se refiere a una falta concreta para un alumno, que tendrá por tanto un argumento de tipo *falta* además del argumento de tipo *alumno,* y otra que se refiera al conjunto

de faltas (de cualquier tipo), para un alumno: así. por ejemplo, tendremos borrar-falta y borrar.

Para las faltas se necesita una operación de igualdad para poder contar cuántas faltas de cierto tipo ha cometido un alumno. El parámetro *FALTAS* es igual al parámetro *ELEM=* (definido en la Sección 1. 1.5). pero con un tipo *falta* en vez de *elemento,* por claridad.

Para generalizar la operación lista-

registrarla. y. registrar! /. a. s)) (conmutatividad)

borrar-falta(/. A. crear)
borrar-falta(/. a. registrar!/, a, s))
borrar-falta(/. a. registrar(g. y, 5))

negra como pide el enunciado añadiremos un argumento más de tipo *nat* que indicara el número de faltas que se han tenido que cometer como mínimo para aparecer en la lista. Las operaciones lista-negra-falta y lista-negra son parciales porque no están definidas cuando ese argumento vale 0.

a. y : *alumno*

f, g : *falta*

n : nat

s ■ fichero-faltas

ecuaciones

registrar! *f,* a. registrarte, y, 5»

crear

borrar-falta(/. a. s)

registrarlg. y. borrar-falta!/, a. s)) <=
frg v
crear
borrarla. 5)
registraría, y. borrada,s)) <= a 7= y

borrarla , crear)
borrarla . registrad/, a. y))
borrarla . registrarla, y, s))

cantidad-falta(/, a. crear) cantidad-	0
falta(/. a, registrad/, a. j))	l + cantidad-falta!/. a, s)
cantidad-falta(/, a. registrarla, y. a))	cantidad-falta!/, a, 5) <= / 7= g V a 7= y

cantidadla. crear) 0
l + cantidadla, s)
cantidadla. .v) <= a 7=
cantidadla.
registrad/, a.s))
cantidad(a. registrar(i>, y, .r))
lista-negra-falta(/. ü, 5) = error
lista-negra-falta(/. *11.* crear) = |] <= n > 0
lista-negra-falta(/, *n,* registrad *f,* a. s))
=insertar-ord(.v. lista-negra-faltaif. *n.* borrar-faltal *f.* a. a)))
<= *11* > 0 A cantidad-faltal/. a. s) > n — 1 lista-negra-falta(/, *11.* registrar!/, a. ,s)) = lista-negra-faltai./. n.s)
<= *11* > 0 A cantidad-faltal/. a. s) < n — l lista-negra-falta!/. *11,* registrar(g. a. s)) = lista-negra-faltal,/. *n.* s) s= «>0 A *J* 7= g

lista-negra(0,5) = error
lista-negra(n. crear) = *[] <= 11 > 0*
lista-negra(ri. registrar(/, x, 5)) =
insertar-ord(x, lista-negra(n. borrar(x, 5)))
<= íɪ > 0 A
cantidadfx, .v) > *n* — 1 lista-
negra(zr. registrar(/. x,.')) =
lista-negra(zz, s)
<= *11* > 0 *A* cantidad(x, j) < *11* — 1
fespecificación

La implementación también puede verse afectada por el número de faltas diferentes. Si este número es pequeño y constante, entonces se puede asociar a cada alumno en la tabla un vector de números naturales indexado por faltas donde cada posición indique cuántas faltas de ese tipo ha cometido el alumno. Sin embargo, si el número de tipos de falta es grande, la idea del vector no es buena, porque podríamos tener muchas posiciones con valor 0. Además, si este número es variable, el vector (estático) puede quedarse pequeño. Para cada alumno, lo que necesitamos es una estructura donde cada tipo de falta esté asociado con el número de faltas de este tipo cometidas por el alumno (siempre que no sea 0). Esta estructura puede ser una tabla donde las faltas se utilicen como claves, y el valor asociado a una clave sea un número natural. Por otra parte, también interesa tener calculado el número total

tipos
info-
fichero-
faltas

de faltas cometidas por un alumno, para poder implementar eficientemente las operaciones cantidad y lista-negra. Por tanto, asociado a cada alumno tendremos un registro con dos campos: un número natural indicando el número total de faltas y una tabla que asocie faltas con números naturales. Así. el tipo representante para el sistema de faltas es:

reg
*núm-faltas : nat**
tabla : tabla[falta, nat]*
freg
tabla[alumno, info-alumnoi

En cuanto a las operaciones, el sistema vacío se consigue creando la tabla vacía. Para registrar una nueva falta, hay que insertar en la tabla asociada al alumno la falta cometida e incrementar el número total de faltas. Como tenemos dos tablas, tenemos dos operaciones combinar, una que combina números naturales (combinar interno) y otra que combina registros de tipo *info-alumno* (combinar externo). El combinar interno corresponde a la suma de números naturales, como en el Apartado (a). El combinar externo tendría que ‵'mezclar” tablas, lo cual no es sencillo. Se puede simplificar el problema si este combinar siempre se queda con el segundo valor. Cuando se inserte en la tabla un valor asociado a un alumno, el valor insertado será ya el modificado, que será el que se quede en la tabla.

proc combinar-interno(n 1 : *nat, en? : nat)*
zij := ri| +no
fproc
proc combinar-externo(í] : *info-alumno,* **c** 12 : *info-alumno)*
il := i2
fproc
proc registrarte *f : falta,* **e** *a : alumno, s : fichero-faltas)*
var r : *info-alumno*
si está-tabla? (a, s) **entonces** *i* := consultar-tabla(«, s) **si no** *i.núm-faltas* := 0 ; *i.tabla* := tabla-vacía() **fsi**
{ i contiene las faltas *ya* cometidas por el alumno *a)*
i.núm-faltas := i.núm-faltas + 1
insertar-tabla(/, 1, *i.tabla)*
insertar-tabla(n, *i, s)*
fproc

Para borrar las faltas de cierto tipo de un alumno, hay que eliminar la información de **dichas faltas** de la tabla interna y decrementar el número total de faltas. Si este número total llega a ser 0. hay **que** eliminar al alumno del sistema, porque solo se mantiene información de alumnos que hayan **cometido** alguna falta.

proc borrar-falta(e *f : falta,* e *a : alumno, s : fichero-faltas')* **var** *i : info-alumno*
si está-tabla?(«, v) **entonces**
i := consultar-tablafa, s)
si está-tabla?(/, *i.tabla)* **entonces**
n := consultar-tabla(/, *i.tabla)* (faltas *f* cometidas)

i.núm-faltas := *i.núm-faltas — n*
si *i .núm-faltas* **> 0 entonces**
eliminar-tabla!/, *i.tabla)*
insertar-tabla(o. *i, s)*
si no
eliminar-tablaffl, 5)
fsi
fsi
fsi
fproc

Para borrar todas las faltas de un alumno simplemente se elimina la información del alumno en el sistema.

proc borrarte *a : alumno, s : fichero-faltas)*
eliminar-tabla(rr, s)
fproc

El número de faltas de cierto tipo cometidas por un alumno se consigue consultando la falta en la tabla asociada a dicho alumno, y la cantidad total se tiene en el campo *núm-faltas.*

fun cantidad-falta! / : *falta, a : alumno, s : fichero-faltas)* **dev** *n : nat* **var** *i : info-alumno*
si está-tabla?*(a, s)* **entonces**
i := consultar-tabla(<í, s)
si está-tabla?(/. *i.tabla)* **entonces** *n* := consultar-tabla!/, *i.tabla)*
si no *n* **:= 0**
fsi
si no *n* **■.= 0**
fsi
ffun

fun cantidad!» : *alumno, s : fichero-faltas)* **dev** *n : nat* **var** *i : info-alumno*

si está-tabla?*(a, s)* **entonces**
i := consultar-tabla(«, *s)*
11 := i núm-faltas
si no *n* **:= 0**
fsi
ffun

Para producir la lista negra asociada a una falta procedemos de forma similar al Apartado (a). Primero se genera una lista ordenada por alumnos recorriendo la tabla principal. Después esta lista se tiltra extrayendo los alumnos con más faltas. De igual forma se obtiene la lista negra que contempla todas las taitas de un alumno en conjunto.

tipos
par-faltas = **reg**
alumno : alumno
info : info-alumno
freg
ftipos
fun lista-negra-falta(/ : *falta, n : naf*'. s : *fichero-faltas)* **dev** *I : lista[alumno]*
var *Ip : lista[par-faltas]*
Ip := recorrer-ordenada(.v) : / := lista-vacía()
mientras ~'es-lista-vacía?(//:>) hacer
x := izquierdo(/p) ; elim-izq(Zp)
si está-tabla?^, *x .info.tabla)* Ac
consultar-tabla(/, *x.info.tabla) > n*
entonces añadir-der(/, *x.alumno)*
fsi
fmientras
ffun
fun lista-negra(íi : *nat*r, r : *fichero-faltas)* **dev** *I : lista[alumnoi*
var *Ip : lista[par-faltas]*
Ip := recorrer-ordenada(s) ; / := lista-vacía()
mientras -'es-lista-vacia?(/p) **hacer**
x := izquierdo(/p) ; elim-izq(/p)
si *x.info.núm-faltas > n* **entonces** añadir-der(/. *x.alumno)* **fsi**
fmientras ffun

Suponiendo que la tabla de alumnos se implementa mediante un árbol de búsqueda equilibrado, y las tablas de faltas de cada alumno se implementan mediante tablas dispersas con accesos y modificaciones de tiempo constante (ya

que las faltas no tienen por qué estar ordenadas), las operaciones tienen el siguiente coste. La operación creares constante. Las operaciones registrar, borrar-falta, borrar, cantidad-falta y cantidad tienen un coste logarítmico respecto al número de alumnos. Las operaciones lista-negra-falta y lista-negra tienen coste lineal respecto al número de alumnos.

10.2. La dirección del Hospital Central de Fanfanisflán quiere informatizar su consultorio médico por medio de un sistema que permita realizar al menos las siguientes operaciones:

- generar un consultorio vacío, sin ninguna información,
- dar de alta a un nuevo médico que antes no figuraba en el consultorio,
- hacer que un paciente se ponga a la espera para ser atendido por un médico, el cual debe estar dado de alta en el consultorio.
- consultar el paciente a quien le toca el turno para ser atendido por un médico, el cual debe estar dado de alta y debe tener algún paciente que le haya pedido consulta.
- atender al paciente que le toque por parte de un médico que debe estar dado de alta y tener algún paciente que le haya pedido consulta, y
- reconocer si hay o no pacientes a la espera de ser atendidos por un médico, el cual debe estar dado de alta.

(a) Especificar un TAD para este consultorio e implementarlo.

(b) Adaptar el consultorio anterior para que sea adecuado para las consultas de urgencias, donde a cada paciente se le asigna una urgencia cuando llega a la consulta, y cada médico atiende siempre al paciente más urgente.

----------------**Solución**------------------

Apartado (a)

El único requisito que necesitamos sobre

el tipo de los médicos es que disponga de una operación de igualdad. Por tanto, el parámetro *MÉDICOS* será igual a *ELEM=* (definido en la Sección 1.1.5) aunque el tipo declarado será *médico* en vez de *elemento,* por claridad. En cuanto a los pacientes, no se requiere ninguna operación, por lo que el parámetro *PACIENTES* será igual a *ELEM* (definido en la Sección 1.1.5) aunque el tipo declarado será en este caso *paciente.*

especificación *CONSULTORIOS[MÉDICOS. PACIENTES]*
usa *BOOLEANOS*
tipos *consultorio*
operaciones
crear —> *consultorio* { constructora }
alta-médico *médico consultorio— consultorio* (constructora)
pedir-consulta *paciente médico consultorio —consultorio* (constructora)
tiene-pacientes?*médico consultorio* —>$_p$ *bool*
sig-paciente *médico consultorio* —»„ *paciente*
atender-consulta*médico consultorio* —>■ $_p$ *consultorio*
operaciones privadas
está[9] *médico consultorio —» bool*
variables
c : consultorio
ni .ni : médico
p. p' paciente

Las operaciones constructoras son crear, alta-médico y pedir-consulta, pero las dos

últimas son *parciales,* dado que no se puede dar de alta a un médico que ya antes ha sido dado de alta en el consultorio i la operación privada está[9] sirve para controlar esta situación), y por otra parte un paciente no puede pedir consulta para un médico que no está dado de alta (la misma operación auxiliar sirve para controlar esta situación por ser la complementaria de la anterior). Sí es posible en cambio que un mismo paciente pida consulta varias veces para un mismo médico.

ecuaciones

alta-médico(m, c)= error <= está?(ni.c)

pedir-consulta(/>. *ni, c)* = error <= -^está?(»i. c)

Para simplificar la presentación de la especificación, a partir de ahora escribimos las ecuaciones suponiendo que todos los términos que involucran constructoras parciales están bien definidos.

Las operaciones constructoras *no son libres,* puesto que:

- El orden en que se dan de alta dos médicos diferentes no importa.
- La operación de pedir consulta para un médico conmuta con la operación de dar de alta otro médico.
- El orden en que se piden las consultas para médicos diferentes no importa; en cambio, sí tiene importancia para el mismo médico, ya que los pacientes se atienden por orden de llegada.

alta-médico(m alta-médico(m', c)) =

alta-médicoun , alta-médico(»n. c))
<= *ni* 5= *ni'*
pedir-consulta(/>. *ni.* alta-médico(»i', c)) = alta-médico(m'. pedir-consultatp. *ni.* <■*))*
<= *ni* = *ni'*
pedir-consulta(/>. *ni,* pedir-consulta(//. *ni', <•))* = pedir-consulta(/>'. *ni .* pedir-consulta</>. *ni.* c))
<= *ni ni'*

Las restantes operaciones se definen distinguiendo en general tres casos según las constructoras.

está? (m, crear) = falso
está?(ni. alta-médico(m', c)) = *m* == *ni' v* está?*(ni*. c)
está?(m. pedir-consulta(p, *ni',* c)) = está?(/n.c)
tiene-pacientes?(ni. crear) = error
tiene-pacientes? *(ni.* alta-médico *(ni.* c)) = falso
tiene-pacientes?*(m.* aita-médico(m', c)) =tiene-pacientes?*(ni. c)*
<= *ni ni*
tiene-pacientes?(ni, pedir-consulta(p, *ni.* c)) = cierto
tiene-pacientes?(ni. pedir-consultafp, ni', c)) =tiene-pacientes?(ni, c)
<= ni ni'
sig-paciente(m, crear) = error
sig-paciente(m, alta-médico(ni, c)) = error
sig-paciente(ni, alta-médico(ni', c)) = sig-pacientefm, c) *<=ni j=.ni'*
sig-paciente(ni, pedir-consulta(p, *ni.* c)) = *p* <=-'tiene-pacientes?*(ni.*

<•)
sig-paciente(ni, pedir-consultafp, *ni. c))*
= sig-paciente(m. c) <= tiene-pacientes?*(ni. c)*
sig-paciente(ni, pedir-consulta(p, *ni.* c)) = sig-paciente(ni. c) <= *ni ni*
atender-consultafm, crear) = error
atender-consulta (ni, alta-médico(ni. c))
= error
atender-consulta (ni. alta-médico(ni', c))
=alta-médico(ni'. atender-consultafm, c))
<= m *ni*
atender-consulta(m, pedir-consulta(p, *ni,* c)) = *c* <= ^tiene-pacientes[9]*(ni.* c)
atender-consulta(m. pedir-consulta(p, ni, c)) = pedir-consultafp. *ni.* atender-consultafm. c))
<= tiene-pacientes?*(ni.* c)
atender-consultafm. pedir-consultafp. *ni,* c)) = pedir-consultafp. m' atender-consultafm. <?))
<= ni *ni*

fespecificación

Un consultorio puede representarse mediante una *labia* cuyas claves son los médicos, a cada uno de los cuales se le asocia una *cola* de pacientes.

tipos

consultorio = tabla[médico, cola[paciente]i

ftipos

La operación crear corresponde directamente a la tabla vacía. Para implementar alta-médicofm, c), primero

buscamos al médico ni en la tabla que representa al consultorio *c.* Si ya está, se trata de un error, y si no, tenemos que insertar la *clave ni* asociada a la cola vacía, ya que el médico m no tiene pacientes inicialmente.

proc alta-médicofe m : *médico, c : consultorio)*
 si está-tabla?(m, *c)* entonces error(El médico ya existe)
 si no insertar-tabla(m. cola-vacia(), c)
 fsi
fproc

El predicado está?(ni.c) corresponde exactamente a buscar m en la tabla c. Para implementar la operación pedir-consultafp, ni. *c),* primero hay que buscar *ni* en la tabla *c.* Si no está, es un error, mientras que si ya está, hay que añadir p a la cola de pacientes asociada a m. De esto puede encargarse la operación combinar entre colas de pacientes. Como la cola de pacientes asociada a un médico también se modifica, y de diferente manera, cuando el médico atiende la consulta, combinar tiene que distinguir los dos casos: si la cola recibida como segundo argumento no es vacía, entonces se añadirá su único elemento (un paciente) a la cola recibida como primer argumento: en otro caso, se avanzará en dicha cola.

proc combinarfci : *cola[paciente], e ci : cola[paciente])*
si -’es-cola-vacía?(c2) **entonces** pedir-vez(c), primero(ci))
si no avanzar(ci)
fsi
fproc
proc pedir-consultafe *p : paciente,* **e** *m : médico, c : consultorio)* **var** *cp : cola[paciente]*
si -■está-tabla?*(in. c)* **entonces** error(El médico no existe)
si no
cp := cola-vacíaO
pedir-vez(cp, *p)*
insertar-tabla (ni, *cp, c)*
fsi
fproc

La implementación del predicado tiene-pacientes?(m, c) busca *m* en *c.* Si no está, es un error, y si está simplemente hay que comprobar si la cola asociada es vacía o no.

fun tiene-pacientes?*(m : médico, c : consultorio)* **dev** *b : bool*
si -■está-tabla? (zn, *c)* **entonces** error(El médico no existe)
si no
b := es-cola-vacía?(consultar-tabla(m, c))
fsi ffun

La implementación de la operación sig-paciente(zzi, c) busca *m* en c. Si no está o está pero la cola asociada es vacía, se trata de situaciones erróneas. En caso contrario, el resultado es el primer

elemento de la correspondiente cola no vacía.

Finalmente, las situaciones erróneas de la operación atender-consulta(zzi. *c)* son las mismas que las de sig-paciente(m. c). En el caso bien definido, hay que quitar el primer elemento de la correspondiente cola no vacía, lo que se consigue insertando en la tabla al médico *m* asociado a una cola vacía.

proc atender-consulta(e *p : paciente, e m : médico, c : consultorio)*
 si -■está-tabla?(zzz. *c)* entonces error(El médico no existe)
 si no
  si es-cola-vacia?(consultar-tabla(zzz, c)) **entonces** error(El médico no tiene pacientes)
  si no
   insertar-tabla(zzz. cola-vacía(), c)
  fsi
 fsi
fproc

Suponiendo que la tabla de médicos se implementa mediante una tabla dispersa (ya que los médicos no tienen que mantenerse ordenados), y que el coste de las operaciones de las colas es constante, el coste de todas las operaciones es constante.

Apartado (b)---------------------

Si los pacientes no se atienden por orden de llegada, sino que se atienden según su *urgencia,* entonces necesitamos una operación para comparar pacientes y

saber cuál es el más urgente. Utilizamos una operación de orden total < entre pacientes, de forma que cuanto "menor" sea el paciente mayor será su urgencia. Por eso en este apartado el parámetro *URGENCIA* es similar al parámetro *ELEM*< definido en el Ejercicio 5.7.

En cuanto a la relación de equivalencia entre términos construidos, hay que indicar que ahora el orden de llegada de los pacientes, aunque sea para el mismo médico, no es importante. La modificación consiste en eliminar la condición de la última ecuación de equivalencia vista en el apartado anterior:

pedir-consulta(p. *ni.* pedir-consulta(p'. *ni.* c)) = pedir-consulta(//. *ni.* pedir-consultafp. *ni,* c))

La especificación de las operaciones está? y tiene-pacientes? no se modifica. Pero sí son diferentes las operaciones sig-paciente y atender-consulta porque ahora hay que tener en cuenta la urgencia de los pacientes, es decir, su relación de orden. El tratamiento es similar al de los elementos en una cola con prioridad, solo que aquí los pacientes se relacionan entre sí solamente si han pedido consulta al mismo médico.

sig-paciente(m. crear) = error

sig-paciente(m. alta-médico(i». <■))= error

sig-paciente (m, alta-médico(m', c)) = sig-paciente(zn, c)

<= ni / ni'

sig-paciente(m. pedir-consultaf/i, *ni. c))*
= *P* <=-■tiene-pacientes?*(ni,*
c)
sig-paciente(zzt, pedir-consulta(p. *ni,*
c)) = *p*
<= tiene-pacientes?*(ni,* c) A
sig-pacienteízzz, *c)* > *p* sig-pacienteízzi,
pedir-consulta(p, *ni.* c)) = sig-
paciente(zzz. c)
<= tiene-pacientes? (zzt. *c)*
A sig-paciente(zzz. *c)* < *p* sig-paciente
(zzt. pedir-consulta(p, *ni,* c)) = sig-
paciente(zzz, c)
<=ni / ni
atender-consulta(zzz. crear) = error
atender-consulta(m. alta-médico(/n. c))
=error
atender-consulta (zzz, alta-médico(m',
c)) = alta-médico(zzz',
atender-consultaízzz. c))
<= *ni / ni*
atender-consulta(m. pedir-consulta(p,
m, c)) = *c* <= -■tiene-
pacientes?(zzz. c)
atender-consulta(zzi, pedir-consultafp.
ni. c)) = *c*
<= tiene-pacientes?(zzz, c) A
sig-paciente*(ni. c)* > *p* atender-consulta
(ni. pedir-consulta (p. *ni,* c)) = pedir-
consultaf *p. ni.* atender-consultafzzz, c))
<= tiene-pacientes?*(ni,* c) A sig-
paciente (zzz. c) < *p* atender-
consultafzzz, pedir-consultafp, *ni,* c)) =
pedir-consulta(p. *ni.* atender-
consultafzzz, c))
<= *ni ni*

En la implementación también hay que tener en cuenta la urgencia de los pacientes. La modificación consiste en tener una *cola con prioridad* asociada a cada médico, en vez de una cola ordinaria.

tipos
consultorio = tabla[médico, colapr[paciente]]
ftipos

La implementación de todas las operaciones es muy similar. Solo se modifican las operaciones sig- paciente y atender-consulta que en vez de utilizar las operaciones primero y avanzar de las colas ordinarias, utilizan las operaciones mínimo y eliminar-mín de las colas con prioridad, respectivamente.

Si la tabla de médicos se implementa mediante una tabla dispersa y las colas de prioridad se implementan mediante montículos, el coste de todas las operaciones es constante, salvo pedir-consulta y atender-consulta, cuyo coste es proporcional al logaritmo del número máximo de pacientes que tiene un médico.

La compañía aérea *Avería* necesita un sistema de reservas en **los vuelos que incluya al menos las** siguientes operaciones:

- crear un sistema de reservas vacío, sin información.
- añadir un *nuevo* vuelo al sistema de reservas,
- comprobar si un vuelo existe o no en el sistema,
- registrar una reserva para un pasajero en un vuelo ya existente, con la condición de que un mismo pasajero no puede hacer más de una reserva en un mismo vuelo.
- comprobar si un vuelo ya no admite más reservas.
- determinar el número de asiento asignado a un pasajero en un vuelo, suponiendo que tal asignación se realiza por orden de reserva empezando por el uno. y devolviendo cero para los pasajeros sin reserva,
- calcular el número de reservas realizadas en un vuelo.
- anular un vuelo en el sistema de reservas, al mismo tiempo que se eliminan todas las posibles reservas para ese vuelo, y
- obtener una lista de todos los vuelos en orden de hora de salida.

Especificar un TAD para el sistema de reservas de *Averia* e implementarlo.

-------------------Solución----------------

Vamos a especificar el sistema de reservas *AVERIA* parametrizado sobre el

tipo de los vuelos y de los pasajeros. Sobre los pasajeros necesitamos una operación de igualdad, ya que se realizan búsquedas por pasajeros, por lo que el parámetro *PASAJEROS* será similar a *ELEM=* (definido en la Sección 1.1.5) aunque con un tipo *pasajero.* Para los vuelos necesitamos una operación que calcule su capacidad para poder determinar si el vuelo está lleno. También necesitamos una operación que devuelva la hora de salida del vuelo, para poder ordenar los vuelos por esta característica. Suponiendo un parámetro *HORAS* similar a *ELEM*< (Ejercicio 5.7) con un tipo *hora* y una operación de orden total < sobre las horas, el parámetro *VUELOS* se especifica a su vez de forma paramétrica:

parámetro *VUELOSjHORAS]*
usa *NATURALES*
tipos *vuelo*
operaciones
  capacidad *vuelo —» nat*
  hora-salida *vuelo —» hora*
fparámetro

Muchas de las operaciones se declaran como *parciales,* ya que su definición depende de si el vuelo al que se refieren está o no incluido en el sistema. La operación enumerar devuelve una lista de suelos ordenada por hora de salida. En vez de utilizar las listas ordenadas especificadas en el Ejercicio 5.7 que requieren una operación de orden entre los elementos que forman las listas, vamos a utilizar

una operación auxiliar que inserta un vuelo en una lista ordenada de vuelos teniendo en cuenta la *hora de salida.* La definición de la operación será muy similar a la homónima del Ejercicio 5.7. pero adaptada a esta situación.

especificación *AVERIAjVUELOS, PASAJEROS]*

usa *BOOLEANOS. NATURALES. LISTAS[VUELOS]*

tipos *averia*

operaciones

crear —> *averia* { constructora)

añadir *vuelo averia* —* (>*averia* { constructora)

existe? *vuelo averia* —* *bool*

reservar *pasajero vuelo averia* — **averia* (constructora]

lleno? *: vuelo avería —>bool*
asiento *:pasajero vuelo avería —>*
p *nat*
núm-reservas : *vuelo avería —>nat*
cancelar *:vuelo avería —> p avería*
enumerar *:avería—» listajvuelo]*
operaciones privadas
insertar-ord : *vuelo listajvuelo] —>*
listajvuelo] **variables**
s : *avería*
v, v' : vuelo
p, p' : pasajero
I : listajvuelo]

Las operaciones constructoras son crear, añadir y reservar, pero las dos últimas son *parciales,* dado que no se puede añadir dos veces el mismo vuelo (la operación existe? sirve para controlar esta situación), y por otra parte un pasajero no puede reservar asiento en un vuelo que no existe o que está lleno o donde ya tiene hecha una reserva (para controlar estas situaciones tenemos las operaciones existe? y lleno?, junto con la operación asiento que devuelve un valor distinto de cero cuando un pasajero ya tiene hecha una reserva).

ecuaciones
añadir(v.s) = error <= existe?(v,s)
reservar!/?, u, s) = error <= -
▪existe?(v. s) v lleno?(u.v) v asiento!/?,
r. s) 0

Para simplificar la presentación de la especificación escribimos todas las ecuaciones restantes suponiendo que

todos los términos que involucran constructoras parciales están bien definidos.

Las operaciones constructoras *no son libres.* Las ecuaciones de equivalencia entre términos construidos indican la conmutatividad que existe entre todas las actividades del sistema de reservas:

- El orden en que se añaden dos vuelos diferentes no importa.
- La operación de hacer una reserva para un vuelo conmuta con la operación de añadir otro vuelo.
- El orden en que se hacen reservas para vuelos diferentes tampoco importa (en cambio, sí tiene importancia para el mismo vuelo).

añadir(t?'. añadir(í), s)) = añadir(t?. añadir(i?'. .s)) <= *v v'*

reservad/?, i?'. añadir(v, s)) = añadirft?, reservar!/?. *v', s)) <= v r'*

reservar!/?'. v'. reservad/?. *v.* s)) = reservar(/?. *v.* reservar!/?', i?'. s)) <= r / v'

Las demás operaciones se definen distinguiendo casos sobre las constructoras. Las operaciones existe? y enumerar son totales, mientras que las restantes son parciales, puesto que no están definidas para un vuelo que no existe en el sistema de reservas.

existe?(u. crear) = falso

existe?(i?, añadiría'. 5)) = u == *v'* v existe?(u.s)

existe?(v. reservad/?. 1?'. s)) =

existe?(u, 5)

La definición de lleno? depende del porcentaje de sobreventa *(overbooking)* permitido. La ecuación adoptada en esta solución no admite sobrevenía, por lo que un vuelo está lleno cuando el número de reservas es igual a la capacidad del vuelo.

Ileno?(u,5) = error <= —'existe?(v, 5)

lleno?(t?.s) = núm-reservas(v, s) == capacidad(u) 4= existe?(i?, s)

Para determinar el número de asiento de un pasajero se busca su reserva. Si no se encuentra una reserva para el pasajero dado, se llega a la operación añadir para el vuelo dado y se devuelve 0 como pide

el enunciado; cuando se encuentra la reserva se utiliza la operación núm-reservas para calcular el número de asiento.

asiento (*p,* u. crear) = error

asiento!/», u. añadirlo, s)) = 0

asiento!/». i>. añadirlo', s)) =
asiento!/», *v,* j) <= v 5¿ r'

asiento!/», o, reservar!/», o, s)) = núm-reservas! o, 5) + I

asiento!/», o. reservar!/»', *v', s)) -*
asiento!/», **o.s) <= p^p'vr^r'**

núm-reservas(o, crear) = error

núm-reservas(o, añadirlo. 5)) = 0

núm-reservas(1», añadirlo'. .?)) = núm-reservas(i>, 5) <= i> 5= *v'*

núm-reservas(u, reservar!/», u, s)) =
núm-reservas! t>. *s)* + I

núm-reservas(o, reservar!/», *v'. s))* =
núm-reservas! *v. s)* <=
v jí v'

La operación cancelar elimina toda la información en el sistema asociada a un vuelo dado, que exista en el sistema.

cancelarlo, crear) = error

cancelarlo, añadirlo, s)) = 5

cancelarlo, añadirlo', s))= añadirlo', cancelarlo. 5)) <= *v jí v'*

cancelar! o. reservar!/», o, ,s)) = cancelarlo, 5)

cancelarlo, reservar!/», o'. 5)) = reservar!/», o', cancelarlo,5)) <= o v'

Para producir la lista de vuelos ordenada por hora de salida se utiliza la operación

privada insertar-ord.
  enumerar(crear) = []
  enumerar!añadirlo. 5)) = insertar-ord(r. enumerar(s))
  enumerar! reservar!/», o. s)) = enumerar(s)
  insertar-ord(o. [ ])   = [o]
  insertar-ord(u. o' /) = o:(u';/) <= hora-salida!o) < hora-salida!o')
  insertar-ordlo. o''/) = 1»': insertar-ordfi»./) <= hora-salida!v) > hora-salida!o') **fespccificación**

Para implementar el sistema de reservas de *Averia* tenemos que tener en cuenta que muchas de las operaciones requieren búsquedas por vuelos. Podemos utilizar una *tabla ordenada* donde las claves sean los vuelos. La información asociada a un vuelo consiste en los pasajeros que tienen una reserva en él. Los pasajeros tienen que estar ordenados según hicieron la reserva. Además tenemos que realizar búsquedas para saber si un pasajero tiene ya una reserva, y para saber la posición que ocupa. Podemos utilizar las *listas* extendidas con operaciones de búsqueda especificadas en el Ejercicio 5.2.

tipos
  averia = tabla[vuelo, lista[pasaiero]]
ftipos

Crear el sistema vacío corresponde a crear la tabla vacía. Para añadir un vuelo primero hay que comprobar que no esté, en cuyo caso se añade a la tabla junto con una lista vacía como valor asociado.

Comprobar si un vuelo existe corresponde simplemente a buscarlo en la tabla. Para registrar una reserva hay que comprobar que el vuelo exista, que no esté lleno y que el pasajero 110 tenga ya una reserva en ese vuelo. En ese caso, se inserta en la tabla el vuelo junto con una lista unitaria que contiene al nuevo pasajero. La operación combinar se encarga de añadir por la derecha a la lista que haya en la tabla asociada al vuelo este nuevo pasajero.

proc combinar!/1 *: lista[pasajero].* **e** 6 : *lista[pasaiero])*
  añadir-der(/|. izquierdo!/»))
fproc

crear	0(1)
añadir,	0(log
reservar	©(lo

Tabla 10.1: Costes de las operaciones del sistema *Averia.*

proc reservarte *p : pasajero,* **e** u : *vuelo, s : averia)* **var** *I : listajpasajero]*
 si -'está-tabla?(v. s) **entonces** error(El vuelo no existe)
 si no
  I := consultar-tablatv. s)
  si longitud(t) = capacidad(u) **entonces** errorfEl vuelo está lleno)
  si no
   si está?(p./) **entonces** error(El pasajero ya tiene reserva)
   si no
    insertar-tablafn, unitaria(p), *s)*
   fsi
  fsi
 fsi
fproc

Para determinar si un vuelo está lleno, primero hay que comprobar si está en la tabla, en cuyo caso hay que comparar la longitud de la lista de pasajeros asociada con la capacidad del vuelo, como se ha hecho en reservar. Calcular el número de asiento de un pasajero consiste en comprobar si el vuelo está, en cuyo caso hay que calcular la posición que el pasajero ocupa en la lista asociada al vuelo en la tabla con la operación posición de las listas extendidas que estamos utilizando (véase el Ejercicio

5.2) que devuelve 0 si el elemento buscado no está, justo lo que necesitamos aquí. Calcular el número de reservas de un vuelo corresponde a calcular la longitud de la lista de pasajeros asociada a ese Mielo, si el vuelo existe en la tabla. Si no. se produce un error. Anular un vuelo consiste en borrarlo de la tabla La lista de todos los vuelos, ordenados por hora de salida, se obtiene a partir del recorrido ordenado de la tabla, como sigue:

tipos
par-reservas = **reg**
vuelo : vuelo
reservas: listajpasajero]
freg
ftipos
fun enumeraras : *averia)* **dev / :** *listajvuelo]*
var *Ip : listajpar-reservas]*
Ip := recorrer-ordenada(s) ; / := lista-vacíaO
mientras -’es-lista-vacía?(/p) **hacer**
x := izquierdo(/p) ; elim-izq(/p)
añadir-der(/, *x.vuelo)*
fmientras
ffun

(a) Suponiendo que la tabla de vuelos se implemento mediante un árbol de búsqueda equilibrado y que las operaciones de las listas tienen coste constante salvo las búsquedas que tienen un coste lineal respecto al número de elementos, el coste de las

operaciones es el dado en la Tabla 10.1, donde V representa el número de vuelos en el sistema y *P* el número máximo de pasajeros en cualquier vuelo.

Se desea un TAD para *carteleras de espectáculos,* **que representen un sistema informático con** información relativa a los espectáculos ofrecidos al público **por una serie de salas. Cada sala puede** ofrecer varios espectáculos. El TAD debe contener las siguientes operaciones:

- crear una cartelera vacía.
- añadir una sala a la cartelera (esta no se modifica si la sala ya **existía).**
- añadir un espectáculo a la oferta de una sala ya existente (el sistema no se modifica si la sala ya ofrecía ese espectáculo).
- eliminar una sala, junto con sus espectáculos (el sistema no se modifica si la sala no existía).
- eliminar un espectáculo de una sala (el sistema no se modifica si la sala no existía o sí existía pero no ofrecía ese espectáculo).
- mostrar la lista de salas del sistema, ordenada y sin repeticiones, y
- mostrar la lista de espectáculos ofrecidos por una sala (vacía si la sala no existe), ordenada y sin repeticiones.

Especificar este TAD e implementarlo.

(b) Ahora se desea extender el TAD de las carteleras con las siguientes nuevas operaciones:

» mostrar la lisia de espectáculos ofertados por el conjunto de las salas (quizá vacía), ordenada y sin

repeticiones, y

o mostrar la lista de salas que ofrecen un espectáculo (quizá vacía), ordenada y sin repeticiones. Especificar las dos nuevas operaciones en una extensión de la especificación anterior e implementar el TAD completo consiguiendo una implementación eficiente para *todas* las operaciones.

----------------Solución**----------------**

Apartado (a)

La cartelera está paramelrizada sobre los tipos de los espectáculos y las salas. Ambos tipos debei tener una operación de orden total para que podamos producir las lisias ordenadas, y una operación di igualdad para poder realizar búsquedas, por lo que los parámetros *ESPECTÁCULOS* y *SALAS* serán comí el parámetro *ALUMNOS* del Ejercicio 10.1. aunque con tipos *espectáculo* y *sala,* respectivamente. L'lili zainos listas ordenadas (véase el Ejercicio 5.7). tanto de salas como de espectáculos. La única operacióí pareial es la que añade un espectáculo a una sala, que solo está definida cuando la sala ya existe. Por ell< declaramos una operación auxiliar que comprueba si una sala ya existe en la cartelera.

especificación *CARTELERAS[ESPECTÁCULOS, SALAS]*
usa *BOOLEANOS. LISTAS-ORDENADAS[SALAS]. LISTAS-ORDENADAS[ESPECTÁCULOS]* **tipos**

cartelera

operaciones

crear —>*cartelera* (constructora)

añ-sala*sala cartelera*—**cartelera* (constructora |

añ-espe*espectáculo sala cartelera* —> *i,* *cartelera*(constructora)

elim-sala*sala cartelera* —♦ *cartelera*

elim-espe-sala*espectáculo sala cartelera* —>*cartelera*

salas *cartelera*—* *lista[sala]*

oferta*sala cartelera* —* *lista[espectáculo]*

operaciones privadas

existe? *sala cartelera —> bool*

variables

c : cartelera

s. s': sala

e.e' : espectáculo

Las operaciones constructoras son crear, añ-sala y añ-espe. esta última *parcial.*

ecuaciones

añ-espe(e, s, c) = error <= -'existe?(s, c)

Las constructoras *no son libres,* ya que:

- Añadir dos veces la misma sala es como añadir la sala una única vez.
- El orden en que se añaden dos salas diferentes no importa.
- Añadir un espectáculo a una sala conmuta con añadir otra sala.
- Añadir dos veces el mismo espectáculo en una sala es como añadirlo una única vez.
- El orden en que se añaden espectáculos para la misma sala o en salas diferentes tampoco es importante.

añ-sala(s, añ-sala(s, c)) = añ-sala(s, c)

añ-sala(s. añ-sala(s', c)) = añ-sala(s', añ-sala(s, c))

añ-espe(e. *s',* añ-sala(s. c)) = añ-sala(s, añ-espe(e,s'. c)) <= *s s'*

añ-espe(e. s. añ-espe(e,s.c)) = añ-espe(e. s. c)

añ-espe(e'. s'. añ-espe(e, s, c)) = añ-espe(<?, s, añ-espeie', *s',* c))

El resto de las operaciones se define distinguiendo casos según las constructoras, y utilizando las operaciones de igualdad entre salas y entre espectáculos. Hay que notar que no es un error añadir a la cartelera información repetida utilizando las

constructoras, lo que deberá tenerse en cuenta a la hora de eliminar y producir listas sin repeticiones.

existe?(s, crear) = falso

existe?(s, añ-sala(s', c)) = *s* == s^1 *V* existe?(s, *c)*

existe?(s. añ-espe(e, *s',* c)) = existe?(s, c)

elim-sala(s, crear) = crear

elim-sala(s, añ-sala(s, c)) = elim-sala(s, c)

elim-sala(s. añ-sala(s', c)) = añ-sala(s', elim-sala(s, c)) <=s/s'

elim-sala(s, añ-espe(e. 5, c)) = elim-sala(s, c)

elim-sala(s. añ-espe(e, *s', c))* — añ-espe(e, 5', elim-sala(s, <’)) 4= *s s'*

elim-espe-sala((?. *s,* crear)= crear

elim-espe-salafe, 5, añ-sala(s', c)) = añ-sala(s', elim-espe-sala(e,s, c))

elim-espe-sala(e, 5. añ-espe(e, *s, c))*

= elim-espe-sala(<?. s, c)

elim-espe-sala(e, s, añ-espe(e', s', c))

= añ-espe(e', *s',* elim-espe-sala(<?. s. c))

<= *e e'* v s *s'*

Para producir las listas ordenadas utilizamos la operación insertar-ord especificada en el Ejercicio 5.7 Obsérvese que tenemos esta operación *sobrecargada* ya que se utiliza para producir tanto listas ordenadas de salas como listas ordenadas de espectáculos. Para evitar repeticiones, las llamadas recursivas se realizan sin el elemento

añadido con insertar-ord.
salas(crear) = [ ]
salas(añ-sala(s, c)) = insertar-ordfs, salas(elim-sala(s,c)))
salas(añ-espe(e, *s,* c)) = salas(c)
ofertáis, crear) = [ ]
ofertáis, añ-sala(s', c)) = ofertáis, c)
ofertáis, añ-espe(<?, s, c))= insertar-ordfe. ofertáis, elim-espe-sala(e, s, c)))
ofertáis, añ-espe(e, s', c)) = ofertáis, elim-espe-sala(e, s, c)) <= s s'
(especificación
Veamos cuál es la representación más adecuada para implementar de forma eficiente las operaciones de la cartelera. Para añadir un espectáculo a una sala, eliminarlo, o consultar la oferta de una sala interesa tener acceso eficiente a la información asociada a una sala. Podemos implementar eficientemente estas operaciones si utilizamos una *tabla ordenada* (implcmentada mediante un árbol de búsqueda de pares, véase el Ejercicio 7.8) donde las claves sean las salas y el valor asociado sea un árbol de búsqueda de espectáculos. Así, el tipo representante para la cartelera de espectáculos es el siguiente:
tipos
cartelera = tablafsala, árbol-bb[espectáculoi]
ftipos
La operación crear consiste en inicializar la tabla como vacía. Para añadir una nueva sala, primero hay que comprobar

si la sala ya aparece en la cartelera. Si es así. la tabla no se modifica. Si no, se inserta en la tabla la sala asociada al árbol de espectáculos vacío. Como sobre el árbol de espectáculos asociado a una sala vamos a necesitar hacer diferentes modificaciones (añadir y eliminar espectáculos), vamos a hacer que la operación combinar utilizada por la tabla se quede siempre con el segundo argumento, último árbol insertado, y que este árbol se inserte asociado a una sala ya modificado.

proc combmar(«i : *árbol-bbfespectáculo], e ai : árbol-bb[espectáculo])*
  «i *:= ai*
fproc
proc añ-sala(e 5 *sala, c : cartelera)*
  si -'está-tabla?(s. *c)* **entonces**
    insertar-tabla(.s. abb-vacío(), c)
  fsi
fproc

Para añadir un espectáculo a una sala primero se comprueba que la sala exista. En ese caso, hay que añadir el espectáculo en el árbol de búsqueda asociado a la sala en la cartelera.

proc añ-espe(e *e : espectáculo,* **e** .r : *sala, c : cartelera)* **var** *ae : árbol-bb[espectáculo]*
  si ^está-tabla?(x, c) **entonces** error(La sala no existe)
  si no
    ae := consultar-tabla(s, *c)*

insertarle, *ae)*
insertar-tablaf.s-, *ae. c)*
fsi
fproc

Eliminar un espectáculo de una sala es la operación "inversa" a la anterior.

proc elim-espe-sala(e *e : espectáculo,* **e** s : *sala, c : cartelera)*
var *ae : árbol-bb[espectáculoi*
si está-tabla?(s, c) **entonces**
ae := consultar-tabla(s, c)
eliminarte, *ae)*
insertar-tabla(.r, *ae, c)*
fsi
fproc

Eliminar completamente una sala consiste en eliminarla de la tabla utilizando directamente eliminar- tabla.

La lista de todas las salas se obtiene recorriendo de forma ordenada la tabla que representa la cartelera, y filtrando la información que nos interesa, que es solo las salas.

crear	$\Theta(1)$
añ-sala, elim-sala	$\Theta(\log S)$
añ-espec, añ-espec-sala	$\Theta(\log S + \log E)$
salas	$\Theta(S)$
oferta	$\Theta(\log S + E)$

Tabla 10.2: Costes de las operaciones de la cartelera.

tipos
par-salas **= reg**
sala : sala
espectáculos : árbol-bb[espectáculo]
frcg
ftipos
fun salas(c : *cartelera)* **dev** *l : lista[sala]*

var *Ip : lista[par-salas]*
Ip := recorrer-ordenada(c) ; / lista-vacía()
mientras -’es-lista-vacía?(/p) **hacer**
.v := izquierdo(/p) ; elim-izq(/p)
añadir-der(/, *x.sala)*
fmientras ffun

La lista ordenada de espectáculos de una sala se obtiene recorriendo en *inorden* el árbol de búsqueda asociado a la sala.

fun ofertáis : *sala. c : cartelera)* **dev / :** *lista[espectáculoi*
var *ae : árbol-bb[espectáculo]*
si está-tabla?(s. *c)* **entonces**
ae := consultar-tabla(s, c)
/ := inorden(ne)
si no / := lista-vacía()
fsi ffun

Si la tabla de salas se implementa mediante un árbol de búsqueda equilibrado y los árboles de búsqueda de espectáculos también son equilibrados, el coste de las operaciones es el dado en la Tabla 10.2, donde *S* representa el número de salas y £ el número máximo de espectáculos en cualquier sala.

Apartado (b)--------------

Extendemos la especificación *CARTELERAS* con las dos nuevas operaciones. Utilizamos una operación privada para eliminar completamente un espectáculo de todas las salas donde se oferta.

especificación *CARTELERAS+[ESPECTÁCULOS, SALAS]*
usa *CARTELERAS/ESPECTÁCULOS. SALAS]*
operaciones
espectáculos : *cartelera —» lista[espectáculo]*
locales *: espectáculo cartelera —> lista[sala]*
operaciones privadas
elim-espe : *espectáculo cartelera —► cartelera*
variables
c : cartelera
s : sala
e.e' : espectáculo

Las listas ordenadas se obtienen de nuevo utilizando la operación insertar-ord especificada **en el Ejer**cicio 5.7. Para evitar repeticiones, en el caso de espectáculos utilizamos la operación auxiliar elim-espe que elimina *completamente* un espectáculo.

ecuaciones
espectáculos(crear) = [ ]
espectáculos) añ-sala(s, c)) = espectáculos(c)
espectáculos(añ-espe(e. s, c)) = insertar-ord(e. espectáculostelim-espete. c)))

elim-espe(e, crear) elim-espe(e, añ-sala(.s, c)) elim-espe(í>. añ-espe(e. s. c)) elim-espe(í'. añ-espe(e'. s, c))

crear
añ-sala(x, elim-espe(e. c»
elim-espe(e, c)
añ-espe(e'. 5. elim-espe(e. c)) <= e =

e’

localesfe. crear)

[]
localesfe, c)
insertar-ord(s. localesfe. elim-sala(s. c)))

localestc. añ-sala(s. <?)) localesíc. añ-espe(e. .s, c))

localesíe. añ-espe(e'. .v. c))

fespecificación

Veamos cuál es la representación más adecuada para implementar de forma eficiente todas las operaciones de la cartelera. Como vimos en el apartado

tipos
carteler

ftipos

anterior, hay operaciones para las que interesa tener acceso eficiente a la información asociada a una sala, lo que se consigue con una tabla ordenada donde las claves sean las salas y la información asociada a cada sala sea los espectáculos que ofrece. Pero para la nueva operación locales es mejor tener acceso eficiente a toda la información asociada a un espectáculo, y para la operación espectáculos interesa tener ordenada la información por espectáculos. Para estas operaciones conviene utilizar otra *tabla ordenada* donde las claves sean los espectáculos, y el valor asociado a un espectáculo sea un árbol de búsqueda de salas. Podemos lograr una implementación eficiente de todas las operaciones si utilizamos las dos tablas comentadas como representación de la cartelera, aunque a costa de duplicar información. Así. el tipo representante para la cartelera de espectáculos es el siguiente:

reg

tabla-salas : tabla[sala, árbol-bb[espectáculoil

tabla-espectáculos : tabla[espectáculo. árbol-bb[sala]]

freg

Las operaciones observadoras consultarán solo una tabla, mientras que las operaciones constructoras y modificadoras actuarán sobre las dos tablas, para mantener la información *consistente.*

La operación crear consiste en inicializar las dos tablas como vacías. Para añadir una nueva sala se opera como en el apartado anterior, aunque ahora se trabaja sobre la primera de las tablas de la cartelera.

Para añadir un espectáculo a una sala hay que añadir el espectáculo en el árbol de búsqueda asociado a la sala en la tabla *tabla-salas* de la cartelera (como antes), pero además, hay que hacer la operación *simétrica* en la tabla *tabla-espectáculos,* añadiendo la sala s en el árbol de búsqueda asociado a *e* en dicha tabla. En este caso la operación combinar también devuelve siempre el segundo argumento.

proc añ-espe(e *e : espectáculo,* **e** 5 : *sala, c : cartelera)* **var** *ae : árbol-bb[espectáculoi. es : árbol-bb[sala]*
si -■está-tabla? (s. *c.tabla-salas)*
entonces error(La sala no existe)
si no
ae := consultar-tablafs. *c.tabla-salas)*
insertare, *ae)*
insertar-tabla(s. *ae, c.tabla-salas)*
as := consultar-tabla(e. *c.tabla-espectáculos)*
insertará, *as)*
insertar-tablate. *as. c.tabla-espectáculos)*
fsi
fproc

A la hora de eliminar un espectáculo de una sala, y para mantener las tablas consistentes, hay que eliminar la información de ambas tablas. Además, si el árbol de salas asociado a un espectáculo queda vacío significa que el espectáculo no se ofrece en ninguna sala. Según la especificación, un espectáculo únicamente está en la cartelera si se ofrece en alguna sala. Por tanto, en el caso de que el árbol quede vacío, habrá que eliminar el espectáculo de la tabla de espectáculos.

proc elim-espe-sala(e *e : espectáculo,* e s : *sala, c : cartelera)* **var** *ae : árbol-bb[espectáculo], as : árbol-bb[sala]*
si está-tabla?!?, *c.tabla-espectáculos)*
entonces
as := consultar-tablafe, *c.tabla-*

espectáculos) ; eliminarte, *as)*
si es-abb-vacío?(os) **entonces** eliminar-tabla(e, *c .tabla-espectáculos)*
si no insertar-tablate, *as, c.tabla-espectáculos)*
fsi
fsi
si está-tabla?te, *c.tabla-salas)* **entonces**
ae := consultar-tablate. *c.tabla-salas) ;* eliminarte, *ae)*
insertar-tablate, *ae. c.tabla-salas)*
fsi
fproc

Eliminar completamente una sala 5 es ahora algo más complicado. De la tabla *tabla-salas* hay que eliminara, pero para mantener la información consistente también hay que eliminar la salas de los árboles de búsqueda asociados a los espectáculos que ofrecía la sala s, en la tabla *tabla-espectáculos.* Además, si alguno de estos árboles queda vacío, habrá que eliminar el correspondiente espectáculo Los espectáculos se recorren utilizando el recorrido en inorden del árbol de búsqueda de espectáculos asociado a s en la tabla *tabla-salas.*

proc elim-sala(e *s : sala, c : cartelera)*
var *ae : árbol-bb[espectáculo], as : árbol-bb[sala], I : lista[espectáculo]*
si está-tabla? (r, *c.tabla-salas)* **entonces**
ae := consultar-tablate, *c.tabla-salas)*
I := inorden(oe)

mientras -■es-lista-vacía?(/) **hacer**
e := izquierdo!/) : elim-izq(í)
as := consultar-tabla(e, *c.tabla-espectáculos) ;* eliminarte, *as)*
si es-abb-vacío?(«s) **entonces** eliminar-tabla(e, *c.tabla-espectáculos)*
si no insertar-tablate, *as. c.tabla-espectáculos)*
fsi
fmientras
eliminar-tablate, *c.tabla-salas)*
fsi
fproc

La lista de todas las salas y la lista ordenada de espectáculos de una sala se obtienen de igual forma al apartado anterior, pero trabajando ahora con la primera de las tablas, *tabla-salas.*

Las operaciones espectáculos y locales son las operaciones *simétricas* a las operaciones salas y oferta, respectivamente, utilizando *tabla-espectáculos* en vez de *tabla-salas.*

tipos
par-espec **= reg**
espec : espectáculo
salas : árbol-bb[sala]
freg
ftipos
fun espectáculos^' : *cartelera)* **dev / :** *listafespectáculo]*
var *Ip : hsta[par-espec]*
Ip := recorrer-ordenadafc.*tabla-espectáculos) ; I* := lista-vacía()
mientras -■es-lista-vacía?(//>) **hacer**
v := izquierdo!*Ip) ;* elim-izq(/p)
añadir-der(/. \.*espec)*
fmientras ffun
fun localesti' : *espectáculo, c : cartelera)* **dev / :** *lista[salai*
var *as : árbol-bb[sala]*
si está-tabla?(e. *c.tabla-espectáculos)* **entonces**
as ;= consultar-tablaje, *c.tabla-espectáculos)*
I := inorden(íi.v)
si no *I* := lista-vacía()
fsi

ffun

Si las dos tablas se implementan con árboles equilibrados, y los árboles internos se mantienen también equilibrados, el coste de las operaciones es el dado en la Tabla 10.2 excepto para la operación elim-sala que se ha convertido en 0(£ (log *E* + log S)). y para las operaciones nuevas espectáculos y locales que tienen costes en <-)(*E)* y 0(log *E* + S). respectivamente.

10.5. El videoclub *Me lo veo todo* desea gestionar los alquileres de películas mediante un sistema informático. El sistema debe incluir al menos las siguientes operaciones:

- crear un videoclub vacío,
- añadir una copia de una película haciendo posible su alquiler,
- incluir a una persona como nuevo socio del videoclub, suponiendo que no lo es.
- establecer que un socio alquila una copia de una película, suponiendo que hay copias disponibles de la película.
- devolver una copia de una película por parte de un socio que la tema alquilada.
- eliminar todas las copias de una película, siempre que hayan sido devueltas.
- determinar si una persona es socio del videoclub.
- calcular el número de copias existentes de una película (alquiladas o no).
- calcular el número de copias alquiladas de una película, y
- determinar si una persona tiene alquilada una copia de una película.

Especificar un TAD para este videoclub e implementarlo.

-------------------------**Solución:**---------------------------

El sistema de alquileres está parametrízado por las películas y los

socios. Para ambos tipos necesitamos operaciones de igualdad, por lo que los parámetros *PELÍCULAS y SOCIOS* serán similares al parámetro *ELEM=* de la Sección 1.1.5. aunque con tipos *película* y *socio,* respectivamente.

especificación *VIDEOCLUBS[PELÍCULAS, SOCIOS]* **usa** *BOOLEANOS. NATURALES* **tipos** *videoclub* **operaciones**

crear : —► *videoclub* (constructora)

añ-copia *: película videoclub —> videoclub* (constructora)

socio*: socio videoclub —videoclub* {constructora)

alquilar *socio película videoclub — videoclub* {constructora)

devolver *: socio película videoclub —> p videoclub*

borrar*: película videoclub —> it videoclub*

es-socio? *: socio videoclub —> bool*

núm-copias *película videoclub —> nat*

núm-alqui *película videoclub —> nat*

tiene-alquilada? : *socio película videoclub —> bool*

variables

p. p' : película s. s' : socio v : *videoclub*

Las constructoras deben determinar el estado del videoclub en cada momento (socios, películas adquiridas y películas

alquiladas), por lo que van a ser las cuatro operaciones crear, añ-copia. socio y alquilar. Las dos últimas son *parciales* según las condiciones del enunciado.

ecuaciones

sociofr. u) = error <= es-socio?(s, *v)*

alquilar(.s, *p,* u) = error <= --es-socio?(s, *v)* V núm-copias(/>, u) == núm-alqui(p. ir)

Para simplificar la presentación de la especificación escribimos las restantes ecuaciones suponiendo que todos los términos que involucran constructoras parciales están bien definidos.

Las ecuaciones entre constructoras indican la conmutatividad que existe entre la mayoría de las actividades del videoclub:

- El orden en que se añaden las películas no importa.
- El orden en que los socios se dan de alta no importa.
- Dar de alta a un socio y añadir una película permutan.
- El orden en que se alquilan películas no importa.
- Alquilar y añadir una copia de la misma película permutan cuando existen suficientes copias sin alquilar.
- Alquilar y añadir una copia permutan cuando se trata de películas diferentes.
- Alquilar y dar de alta a un socio permutan cuando se trata de socios diferentes.

añ-copia(p, añ-copia(p'. v)) = añ-

copia(//, añ-copia(p, v))
 sociofr, sociofs'. u)) = socio(.?'. sociofs, u))
 añ-copia(/r. socio(r, v)) = socioG?, añ-copia(p, v))
 alquilaría, *p.* alquilare?', p', v))= alquilare?', *p',* alquilare?, *p.* v))
 alquilare?, *p.* añ-copia(p, t>)) = añ-copia(p, alquilare?, *p,* u))
 <= (núm-copiastp, *v)* —
 núm-alqui(p. u)) > 0 alquilare?, p,
 añ-copia(p', u)) =añ-copia(//,
 alquilarf?, *p,* u)) <= *p* / *p'*
 alquilarte, p. socio(?', u)) = socio(?', alquilaría-, *p,* u)) <= .? / *s'*

El resto de operaciones se definen distinguiendo casos sobre las constructoras y utilizando las operaciones de igualdad entre películas y entre socios.

devolveres, *p.* crear) = error

devolveres, *p.* añ-copia(/;'.u)) = añ-copia(/;'. devolveres, *p.* r))

devolveres,/>. sociofs', u)) = socio(s',
devolveres, *p.* i->)

devolverfs. *p.* alquilares, *p,* u)) = v

devolveres, *p.* alquilares', *p'.* t>)) = alquilares', *p'.* devolveres. />. r)> <= *p £ p'* v s # s'

borrare/;, crear) = crear

borrar!/;. añ-copia(/;. u)) = borrad/;, *v)*

borrare/;. añ-copia(/;'. u)) = añ-copia(/;'. borrad/;, u)) <= *p p'*

borrart/;. socio(s, v)) = sociofs, borrad/;, v))

borrar!/;, alquilares,/;, ir)) — error

borrart/;. alquilares./;'. u)) = alquilares./;'. borrar!/;, r)) <= *p r p'*

es-socio?(s. crear) = falso

es-socio’es. añ-copia(/;. v)) = es-socio?(s. v)

es-socio?(s. sociofs', v))= s == s' v es-socio’(s, ir)

es-socio?(s. alquilares',/;, v)) — es-socio?(s.v)

núm-copias(/;. crear) = 0

núm-copias(/;. añ-copia(/;, i;)) =
núm-copias(/;. u) + 1

núm-copias(/;. añ-copia(/;', u)) =
núm-copias(/;. u) <= *p* ±

p

núm-copias(/;, sociofs. u)) = núm-copias(/;. i’)

núm-copias(/;. alquilares./;', u)) =
núm-copias(/;. ir)

núm-alqui(/>. crear)= 0

núm-alqui(/;, añ-copia(/;', u))= núm-alqui(/;.i[1])

núm-alqui(/;, socio(s, v)) = núm-alqui(/;. u)

num-alqui(/;. alquilares. /;. v))= núm-alqui(/z. v) + 1

num-alqui(/;. alquilar(s./>', v)) = núm-alqui(/;, *v) <= P r P*

tiene-alquilada?(s, /;, crear)= falso

tiene-alquilada?(s. /;. añ-copia(/;'.
t>)) =tiene-alquilada?(s. /;. v)

tiene-alquilada?(s, /;. sociofs'. v)) =
tiene-alquilada?(s, /;. v)

tiene-alquilada?(s,/;,alquilares./;,
t>)) =cierto

tiene-alquilada?(s./?. alquilares'./>'. u)) = tiene-alquilada?(s./;, r) s= *p* /;' v s s'

fespecificación

Veamos qué estructura necesitamos para poder implententar eficientemente todas estas operaciones. Tenemos que ser capaces de acceder de forma rápida a la información asociada a una película, para poder saber el número de copias o el número de alquiladas, y para poder borrar una película, por ejemplo. Por tanto, interesa tener una *tabla* donde las claves sean las películas. La información asociada a una película será un registro

con dos campos: el número de copias en total y el número de copias alquiladas. Hay otras operaciones para las cuales es importante poder hacer búsquedas de un socio eficientemente, como para saber si alguien es socio, o para alquilar o devolver una película un socio concreto. Por tanto, también tendremos otra *tabla* donde las claves sean los socios. La información asociada a un socio será el *malticoiijunto* de películas que tiene alquiladas, ya que estas no tienen por que estar ordenadas de ninguna manera. Por otro lado, para ninguna operación es necesario tener el conjunto de socios que tienen alquilada una película dada, por lo que no tendremos explícitamente esa información.

tipos

info-peli = **reg**

núm-copias : naf+

nám-alquilailas : nal

freg

videoclub **= reg**

tabla-pelis : tabla[película, info-peli]

labia-socios : tabla[socio, multiconjunto[película]]

freg

ftipos

Crear el videoclub vacío consiste en inicializar las dos tablas como vacías. Para añadir una copia más de una película, hay que insertar en la tabla de películas el par (1,0). donde la operación

de combinación en esta tabla consistirá en sumar, componente a componente, dos registros *info-peli.*

proc combinar-tabla-pelis(í| : *info-peli.* e *1'2 : info-peli)*
 ii.núm-copias i i .núm-copias + ii-núm-copias
 i] .núm-alquiladas := i\.núm-alquiladas + ii.núm-alquiladas
fproc
proc añ-copia(e *p : película, v : videoclub)*
var *i : info-peli*
 i.núm-copias := 1 ; *i.núm-alquiladas :=* 0
 insertar-tablaíp. *i. v.tabla-pelis)*
fproc

Para dar de alta a un socio hay que insertarlo en la tabla de socios junto al multiconjunto vacío de películas, siempre que no esté. Como el multiconjunto de películas asociado a un socio puede modificarse de varias maneras, es más sencillo hacer que la operación de combinación en la tabla de socios devuelva siempre el segundo argumento, es decir, la información modificada.

proc combinar-tabla-socios (<?i : *multiconjunto[pelicula]. e ci : multiconjuntofpelicula]}*
 ci := C2
fproc
proc socio(e s : *socio, v : videoclub)*
 si está-tabla?(s, *v.tabla-socios)* entonces
  error(El socio ya está dado de alta)

si no insertar-tablaís. cjto-vacío(), *v.tabla-socios)*
fsi
fproc

Para alquilar una película hay que comprobar que el socio esté dado de alta, que la película exista y que haya copias sin alquilar. En ese caso se modifican ambas tablas. Obsérvese que las dos tablas no se tratan de la misma manera. En la tabla de películas solo se inserta el par (0. I), porque la operación de combinación modifica la información almacenada. En cambio, con la tabla de socios hay que consultar la información, modificarla e insertarla ya modificada.

proc alquilar(e *s : socio,* **e** *p : película, v ; videoclub)*
var *c : multiconjunto[película]*
si está-tabla?(í, *v.tabla-socios)* entonces
si está-tabla? (p, *v.tabla-pelis)* entonces
i := consultar-tabla(p, *v.tabla-pelis)*
si *i.núm-copias* > *i.núm-alquiladas* entonces
i.núm-copias := 0 ; *i.núm-alquiladas* := 1
insertar-tablaíp, *i, v.tabla-pelis)*
c := consultar-tabla(s. *v.tabla-socios)*
añadirfp, c)
insertar-tabla(s, c, *v.tabla-socios)*
sino error(No hay copias de la película)
fsi

sino error(La película no existe)
fsi
si no error(El socio no existe)
fsi
fproc

La operación devolver es la inversa de alquilar, por lo que hay que restar 1 a *núm-alquiladas* (insertando un registro *info-peh* con — 1 en el segundo campo), y eliminar la película del multiconjunto de películas del socio. Para eliminar completamente una película que no está alquilada, simplemente hay que eliminar de la tabla de películas la película en cuestión. El hecho de que la película eliminada no esté alquilada es la razón de que no necesitemos saber qué socios tienen alquilada la película, para mantener la información consistente.

La operación es-socio? corresponde a buscaren la tabla de socios. Las operaciones núm-copias y núm- alqui corresponden a consultar en la tabla de películas la información asociada a una película y devolver uno de los campos. Por último, para saber si un socio tiene una película alquilada hay que comprobar si la película está (tiene una multiplicidad mayor que 0) en el multiconjunto de películas asociado al socio.

fun tiene-alquilada?(s : socio, /> : *película,* u : *videoclub)* **dev** *b : bool*
var c : *multiconjunto[película]*
si está-tabla? (r, *v.tabla-socios)*
entonces

c := consultar-tabla(s. *v.rab/a-socios)*
b := multip(/i, *c)* > 0
si no *b :*= falso
fsi
ffun

Ya que la información no tiene que mantenerse ordenada y las tablas no tienen que recorrerse, las tablas pueden estar implementadas mediante tablas dispersas cuyas operaciones suponemos constantes. Si las operaciones sobre los multiconjuntos son constantes, salvo las operaciones multip y quitar que tienen un coste lineal sobre el número de elementos, entonces las operaciones del videoclub son todas constantes, salvo devolver y tiene-alquilada que son lineales respecto al número máximo de películas que tiene alquiladas cualquier socio.

Una empresa de muebles de cocina necesita un tipo abstracto de datos para representar el conjunto de muebles colocados en la pared de una cocina. Una cocina se crea con una longitud positiva, y un mueble colocado en la pared se identifica con el par formado por su posición (es decir, por la distancia desde su lateral izquierdo al extremo izquierdo de la pared) y su anchura (la profundidad y la altura no tienen interés pues son iguales para todos los muebles). El TAD debe soportar las siguientes operaciones:

- crear una cocina con una longitud dada.
- añadir un mueble de una determinada anchura en una posición dada, si el mueble cabe.
- determinar si un mueble de una cierta anchura puede colocarse en una posición dada.
- calcular el número de muebles en la cocina.
- determinar cuál es el primer mueble de la cocina, empezando a contar por la izquierda, suponiendo que no está vacía,
- eliminar el primer mueble de la cocina, suponiendo que existe.
- determinar cuál es el mueble *i*-ésimo de la cocina, empezando a contar por la izquierda, siempre que haya al menos *i* muebles.
- eliminar el mueble í-ésimo de la cocina, si existe, y

- mover el mueble í-ésimo de la cocina (si existe) hacia la izquierda, hasta que se junte con el mueble (í — I)-ésimo o el extremo izquierdo de la pared.
- Especificar dicho TAD e implementarlo.

-----------------Solución---------------

Primero especificamos como parámetro la estructura de los muebles, definidos siguiendo el enunciado como un par formado por la posición y la anchura.

parámetro *MUEBLES*
usa *NATURALES*
tipos *mueble*
operaciones
:natnat —> mueble
{constructora)
pos *:mueble —> nat*
anch *: mueble —> nat*
variables
p. a : nat
ecuaciones
pos((p.a» = *p*
anch((p. n)) = *a*
fparámetro

La signatura de la especificación principal incluye todas las operaciones del enunciado más una operación auxiliar tope que explicaremos más adelante.

especificación *COCINA[MUEBLES]*
usa *NATURALES, BOOLEANOS*
tipos *cocina*
operaciones
crear*: nat —> cocina*{ constructora i
añadir *: mueble cocina —cocina*

{constructora)

cabe? : *cocina mueble —> bool*

núm-muebles: *cocina—> nat*

primero : *cocina—mueble*

eliminar-prim: *cocina—>„ cocina*

mueble-n *nat cocina —mueble*

eliminar-n : *nat cocina —>„ cocina*

mover-n : *nat cocina —cocina*

operaciones privadas

tope : *nat cocina —>■ p nat*

variables

i, l. p. a, p'. a' : nat

m. ni' : mueble

c : cocina

Las constructoras son crear y añadir, siendo esta última *parcial.* Siguiendo las ideas del enunciado, un término de la forma

añadirf (*pi, ai*), añadirf (*pi, ai*),

añadirf (p_2. a_2)), crearf/))),

con $p\backslash < p_2 < P3$. se interpreta como una cocina que sigue el esquema superior de la Figura 10.1.

ecuaciones

añadirfzíi, c) = error <= -'cabe?(m,c)

Para simplificar la presentación de la especificación, escribimos las ecuaciones suponiendo que todos los términos que involucran añadir están bien definidos.

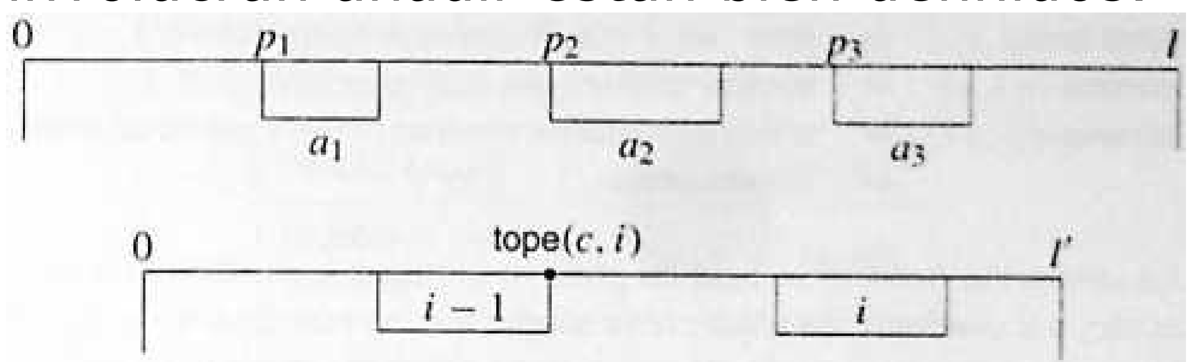


Figura 10.1: Dos configuraciones de muebles

de cocina.

Las constructoras *no son libres,* pues el orden en que se añaden los muebles no importa, siempre qut se dé lugar a una configuración correcta, es decir, que los muebles quepan.

añadir(/i. añadir(m. c)) = añadir(m. añadiría, c))

Las restantes operaciones se definen en términos de las constructoras.

Para determinar si un mueble cabe en la posición dada hay que comprobar que *no* solapa con ningunc de los muebles ya colocados en la cocina. Para ello hace falta conocer la posición y la anchura de los muebles, que se obtienen utilizando la operación {_. _) constructora de muebles. Otra posibilidad sería utilizar las operaciones pos y anch.

cabe?((*p.a).* crearf/)) = *p* + *a<l*

cabe[9]((*p. a).* añadir((*p*'. *a*'). c)) = *((p* + *a<p') v (p'+ a < pY)* A cabe?((*p. a). c)*

num-muebles(crear(/)) = 0

núm-muebles(añadir(m, c)) = núm-muebles(c) + 1

Las operaciones primero y eliminar-prim dan error en el caso vacío, son inmediatas en el caso unitario, y necesitan hacer una comparación cuando hay más de un mueble, para asegurar que el primer mueble es en efecto el que está colocado más a la izquierda.

primero(crear(/)) = error

primero(añadir(m, crear(/))) = *ni*

primero(añadir(m. añadiría;', c)))= *ni*
<= pos(ai) <
pos(primero(añadir(/n'. c)))
primero(añadir(m. añadiría;', c))) =
primeroíañadirím'. c))
<= pos(m) >
pos(primero(añadir(m'. c)>)
eliminar-prim(crear(/)) = error
eliminar-prim(añadir(ai. crear(/))) =
crear(/)
eliminar-prim(añadir(m. añadir(m', c)))
= añadiría;', c)
<= pos(m) <
pos(primero(añadir(m'.
<■)))
eliminar-prim(añadir(m, añadiría;', r)))
= añadirfm. eliminar-
prim(añadir(;;; .<-)))
e= pos(m) >
pos(primero(añadir(m’.
<•)))

Las operaciones mueble-n y eliminar-n distinguen casos según el argumento *i:* si *i* no corresponde a un mueble de la cocina, dan error; para í = 1 se reducen a primero y eliminar-prim respectivamente: y para *i* > I. se hace una llamada recursiva con *i* — 1. utilizando en este caso como operaciones auxiliares primero y eliminar-prim.

mueble-n(í, c) = error <= *i* == 0 v *i* > núm-muebles(c)
mueble-n (1. c) = primero(c) <= núm-muebles(c) > I
mueble-n(i, c) = mueble-n(í — l.

eliminar-prim(c)) <= núm-muebles(e) > *i*
A i > l

eliminar-n(z. c) = error <= *i* == 0 v *i* > núm-muebles(c)

eliminar-n(l. *c)* = eliminar-prim(c) <= núm-muebles(c) > 1

eliminar-n(í, c) = añadir(primero(c), eliminar-n(í — 1, eliminar-prim(c)))

<= núm-muebles(c) > *i* A *i* > 1

La operación mover-n se basa en primero eliminar (con eliminar-n) el mueble z-ésimo de la cocina (si existe) y a continuación añadir (con añadir, que en este caso no puede dar error) un mueble igual en anchura pero con una posición diferente, que es bien la posición del lateral derecho del mueble (z - 1)- ésimo o bien el extremo izquierdo de la pared. Para esto se utiliza la operación auxiliar tope (ilustrada en el esquema inferior de la Figura 10.1) cuya definición se basa simplemente en distinguir esas dos posibilidades, y en la primera sumar la anchura y la posición del mueble (z — 1)-ésimo.

mover-n(z\ c) = error <= *i* == 0 v *i* > núm-muebles(c)

mover-n(i. c) = añadir((tope(í. c). anch(mueble-n(i, c))}. elimmar-n(z. c))

<= núm-muebles(c) > *i A i* > 1

tope(z, c) = error <= *i* == 0 V *i* > núm-muebles(c)

tope(l.c) = 0 <= núm-muebles(c) > 1

tope(z.c) = pos(mueble-n(z - 1. c)) + anch(mueble-n(z - 1, c))

<= núm-muebles(c) > *i* A *i* > I

fespecificación

En cuanto a la implementación conviene utilizar una estructura lineal que represente la secuencia de muebles en la pared. Para que la posición de cada mueble en la estructura coincida con el número del mueble en la pared interesa mantener la estructura ordenada por posiciones en la pared de los muebles, de menor a mayor. Además interesa poder acceder directamente a un mueble sabiendo su número. Por todo ello utilizaremos una *lista ordenada* de muebles (véase el Ejercicio 5.7) y con acceso por posición (véase el Ejercicio 5.8). El orden entre los muebles vendrá dado por las posiciones que ocupan en la pared: un mueble será *menor* que otro si su posición (un número natural) es menor, es decir, el mueble menor está colocado más a la izquierda.

También tenemos que conocer la longitud de la cocina. Por eso el tipo representante será un registro con dos campos: la longitud de la cocina y la lista de muebles. Estos a su vez serán registros con dos campos: la posición del mueble en la pared y su anchura.

```
tipos
mueble = reg
        posición : nat
        anchura : naF
       freg
 cocina = reg
        longitud : nat*
        muebles : lista[mueble]
```

freg
ftipos
Crear una nueva cocina consiste en inicializar el campo *longitud* con la longitud dada y la lista de muebles a vacía. Para añadir un mueble hay que comprobar que cabe, en cuyo caso se inserta de forma ordenada en la lista.

proc añadir(e *ni : mueble, c : cocina)*
si cabe?(/zt.c) **entonces** insertar-ord(zzz, c)
si no error(El mueble no cabe)
fsi
fproc

crear, núm-muebles, primero, elim-prim	$\Theta(1)$
añadir, cabe?	$\Theta(M^2)$
mueble-n, eliminar-n, mover-n	$\Theta(M)$

Tabla 10.3: Costes de las operaciones de la cocina.

Para determinar que un mueble cabe en la cocina hay que comprobar que no solapa con ninguno de los muebles ya colocados. Para ello se recorre la lista de muebles, accediendo con la operación consultar a cada mueble dada su posición en la lista. Nótese que el recorrido puede acabar en cuanto haya solapamiento con algún mueble colocado (con resultado falso) o cuando el nuevo mueble pueda colarse delante del mueble más a la izquierda de los que quedan por considerar (con resultado cierto).

fun cabe[9]**(m :** *mueble, c : cocina)* **dev** *b : bool*

b := cierto . *puede-delante :=* falso

i' — I ; / := longitud(c.nmeWes)

mientras / < / A Z> A -•*puede-delante* **hacer**

ni := consultarle.*muebles, i)*

puede-delante := m .posición + m .anchura < ni.posición

b = puede-delante v *(ni.posición + ni.anchura < m.posición)*

i := i + 1

fmientras ffun

El número de muebles coincide con la longitud de la lista de muebles. Para obtener el primero basta con utilizar la operación consultar con índice 1 o utilizar la operación izquierdo de las listas. El primero se elimina con elim-izq. La operación mueble-n corresponde a la operación consultar y la operación eliminar-n corresponde a la operación eliminar de las listas con acceso por posición (véase el Ejercicio 5.8). Para mover el mueble í-ésimo se modifica la posición i-ésima de la lista con modificar que recibe como argumento el mueble en su nueva posición.

proc mover-n(e / . *nat. c cocina)*

si *i* == 0 v *i* > núm-muebles(c) **entonces** error(El mueble no existe)

si no

ni := mueble-n(L *c)*

si *i* = 1 **entonces** *imposición :=* 0

si no

ni := mueble-n(z — l.c)
m.posición := ni .posición + ni .anchura
fsi
modificarte.*muebles, i. ni)*
fsi
fproc

Suponiendo que las operaciones sobre las listas tienen un coste constante, salvo insertar de forma ordenada y acceder por posición, que suponemos tienen coste lineal respecto al número de elementos en la lista, las operaciones de la cocina tienen los costes dados en la Tabla 10.3. donde *M* es el número de muebles en la cocina.

PARTE 2: MÉTODOS ALGORÍTMICOS

Capítulo 11

11. DIVIDE Y VENCERÁS

ESQUEMA GENERAL

El método ***divide y vencerás*** consiste en descomponer el problema que hay que resolver en una serie de **subproblemas**, resolver estos subproblemas, y después combinar los resultados para obtener la solución del problema original.

Lo importante es que *los subproblemas son del mismo tipo que el problema original*, pero de *menor tamaño*, y *se resuelven usando la misma técnica*. De esta forma, el método se expresa de manera natural mediante un algoritmo **recursivo**, cuyo esquema general se muestra a continuación:

```
fun divide-y-vencerás(x : problema)
dev y : solución
  si pequeño(x) entonces
    y:= método-directo(x)
  si no
    {descomponer x en k ≥ 1 problemas más
    pequeños}
    ⟨x1,…,xk⟩:= descomponer(x)
    {resolver recursivamente los
    subproblemas}
    para j=1 hasta k hacer
```

```
      yj := divide-y-vencerás(xj)
    fpara
    {combinar los yj para obtener una
    solución y para x}
    y := combinar(y1,…,yk)
  fsi
ffun
```

La función pequeño(x) es un predicado que determina si el tamaño del problema x es *suficientemente pequeño* para ser resuelto sin dividir más. Si es ese el caso, el problema se resuelve mediante la función método-directo(x). En otro caso, el problema x se divide en subproblemas $x_1,\dots,x_k$. que son resueltos a su vez mediante divide-y-vencerás. La función combinar($y_1,\dots,y_k$) calcula la solución para x a partir de las soluciones $y1,\dots,y_k$ de los correspondientes subproblemas.

Costes

Para que la aplicación del método *divide y vencerás* merezca la pena debe cumplirse que las operaciones descomponer y combinar sean bastante eficientes, que el número de subproblemas generados sea pequeño, y que los subproblemas sean aproximadamente del mismo tamaño y no solapen entre sí.

Si el problema x es de tamaño *n y* los tamaños de los subproblemas $x_1,\dots,x_k$ son, respectivamente, $n_1,\dots,n_k$ podemos describir el coste en tiempo del algoritmo divide-y-vencerás mediante la siguiente

recurrencia:

$$T(n) = \begin{cases} g(n) & n \le n_0 \\ \sum_{j=1}^{k} T(n_j) + f(n) & n > n_0 \end{cases}$$

donde T(n) es el tiempo del algoritmo divide-y-vencerás cuando el problema de entrada es de tamaño *n, n_0* es el tamaño *umbral* que marca cuándo no se ha de seguir dividiendo, g(n) es el tiempo de método-directo y *f(n)* el tiempo de descomponer en subproblemas y combinar los resultados de estos.

En particular, la complejidad de muchos de los algoritmos divide y vencerás desarrollados en este capítulo corresponde a recurrencias de la forma

$$T(n) = \begin{cases} c & 0 \le n < b \\ aT(n/b) + f(n) & n \ge b \end{cases}$$

donde $a.c \in \mathbb{R}^+$ *y* $n.b \in \mathbb{N}$. Suponiendo además que f(n) ∊ $\Theta(n^k)$ para $k \in R^+ \cup \{0\}$. podemos utilizar la siguiente tabla para calcular la complejidad de *T(n)* [Peñ98, Capítulo I]:

$a < b^k$	$T(n) \in \Theta(n^k)$
$a = b^k$	$T(n) \in \Theta(n^k \log n)$
$a > b^k$	$T(n) \in \Theta(n^{\log_b a})$

Notación

En la mayoría de los ejercicios aquí propuestos, generalmente se trabaja con vectores de cierto tamaño que inicialmente, tienen índices entre 1 y n. Pero al aplicar la técnica de *divide y*

vencerás, se utilizarán vectores de menor tamaño cada vez, que corresponderán a ***secciones*** del vector inicial.

Por eso proponemos la siguiente notación para las soluciones que a continuación se presentan: además del vector inicial V[1..n], las funciones y procedimientos recibirán, como parámetros de entrada, 2 índices, *c y f.* que representan, respectivamente, el *comienzo y final* de la sección de trabajo V[c.. f].

EJERCICIOS RESUELTOS DYV

11.1. Adivinar numero

Dos amigos juegan a **adivinar** un número. Una persona piensa un número natural positivo, y la otra persona debe adivinarlo preguntando solamente si es **menor o igual** que otros números. Diseñar un algoritmo eficiente para adivinar el número.

--------------------Solución:--------------------

Como el número a adivinar puede ser arbitrariamente grande, empezar preguntando por el 1 y seguir después en secuencia (búsqueda lineal) hasta alcanzar el número propuesto no es un método práctico.

En su lugar, necesitamos hacer una ***búsqueda binaria***, que nos permita reducir de forma más rápida el conjunto de candidatos. Sin embargo, el algoritmo de búsqueda binaria funciona con vectores de tamaño fijo y conocido, por lo que lo primero será encontrar alguna **COTA superior** del número a adivinar (en principio, una cota inferior es 1). Para encontrar dicha cota habrá que seguir un método que genere números cada vez más grandes y de modo que el incremento también aumente de forma rápida. A tal efecto, podemos utilizar, por ejemplo, las potencias de 2.

Vamos a recordar primero el algoritmo de **búsqueda binaria en un vector ordenado** (que es un ejemplo típico de algoritmo de tipo *divide y vencerás)*: se compara el elemento a buscar con el

elemento que ocupa la posición central en el vector; si este último es mayor que aquel, entonces habrá que buscar en la primera mitad del vector; mientras que si es menor, se seguirá la búsqueda en la segunda mitad.

```
{V[1] ≤ ... ≤ V[n] ∧ 1≤c≤f+1≤n+1}
fun búsqueda-binaria(V[1..n] de
elemento, e: elemento, c, f : nat) dev
⟨existe : bool, p : nat⟩
  si c > f entonces
    ⟨existe, p⟩ := ⟨falso, c⟩
  si no
    m := (c + f) div 2
    casos
       e<V[m]→⟨existe.p⟩:=búsqueda-
      binaria(V,e,c,m-1)
      □ e=V[m]→⟨existe.p⟩:=(cierto, m}
      □ e>V[m]→(existe.p}:=búsqueda-
      binaria(V,e,m+1,f)
    fcasos
  fsi
ffun
{(existe ⇒ c < p < f ∧ V[p]=e) ∧ (¬existe⇒
c<p<f+1 ∧ V[c..p - 1] < e < V[p..f]) )
```

La condición $c < f$ asegura que el tamaño del problema (la distancia entre f y c) decrece con cada llamada recursiva. La llamada inicial sería búsqueda-binaria(*V*, *e*, 1, *n*).

El número de comparaciones realizadas por búsqueda-binaria viene dado por la siguiente recurrencia:

$$T(n) = \begin{cases} c_0 & n \leq 1 \\ T(n/2) + c_1 & n > 1 \end{cases}$$

donde *n* es la distancia entre *f* y *c* (*n*=f-

c+1) y c_0 y c_1 son constantes reales positivas. Por tanto. $T(n) \in \Theta(\log n)$.

Para adivinar el número utilizaremos una variación de *búsqueda-binaria* donde no necesitamos un vector porque se trabaja con un intervalo de enteros positivos y. además, siempre vamos a tener éxito. Esta nueva versión en lugar de recibir el número a buscar como argumento, lo devuelve como resultado: para ello utiliza una función "interna" menor-igual que devuelve cierto si y solo si el número a adivinar es menor o igual que el propuesto (como argumento).

```
{c ≤ n ≤ f}
fun búsqueda-binaria2(c, f: nat) dev n: nat
  si c=f entonces
    n := c
  si no
    m := (c + f) div 2
    si menor-igual (m) entonces
      n := búsqueda-binaria2(c,m)
    si no
      n := búsqueda-binaria2(m+1, f)
    fsi
  fsi
ffun
```

El algoritmo completo que simula el juego de adivinación de los dos amigos es el siguiente:

```
fun adivinar() dev n: nat
  {encontrar cotas superior e inferior}
  cinf := 1;
  csup := 1;
  mientras ¬ menor-igual(csup) hacer
```

```
    cinf := csup + 1;
    csup := 2 * csup
  fmientras
  n := búsqueda-binaria2(cinf, csup)
ffun
```

Aunque la función no recibe argumento alguno, su resultado depende del número "oculto" a adivinar, el cual determina el comportamiento de la función menor-igual.

Si n es el número a adivinar, para determinar el intervalo donde hacer la búsqueda binaria se necesitan $m = \lfloor \log n \rfloor + 1$ preguntas, y el tamaño del intervalo donde hacer la búsqueda binaria es 2^m. Por tanto, la cantidad total de preguntas a realizar, en el caso peor, está en Θ(log n). Podríamos considerar otra base para las potencias, observando que cuanto mayor sea dicha base antes se encontrará la cola superior. Sin embargo, hay que tener en cuenta que, en general, el intervalo que quedará para la búsqueda binaria será también mayor.

CHATGPT

El pseudocódigo presentado describe un algoritmo para adivinar un número natural positivo n mediante una variante de la búsqueda binaria adaptada a un contexto donde no se dispone de una lista o vector de elementos, sino de un rango de números enteros positivos dentro del cual se encuentra el número a adivinar. Este algoritmo se basa en la estrategia de dividir el espacio de búsqueda a la mitad en cada paso, reduciendo significativamente el número de preguntas necesarias para llegar al número correcto. A continuación, detallaré el funcionamiento de las dos funciones principales que componen este algoritmo: `búsqueda-binaria2` y `adivinar`.

Función `búsqueda-binaria2(c, f: nat) dev n: nat`
Esta función implementa la búsqueda binaria adaptada y tiene dos parámetros: c (cota inferior del rango de búsqueda) y f (cota superior del rango de búsqueda). La función devuelve el número n, que es el número a adivinar.
1. **Caso Base**: Si c = f, significa que hemos reducido el rango a un único posible valor, por lo que este debe ser el número a adivinar. La función entonces asigna n := c y termina.
2. **Caso Recursivo**: Si c ≠ f, el algoritmo calcula el punto medio m del rango [c, f] como (c + f) div 2. Luego, utiliza la función menor-igual(m) para preguntar si el número a adivinar es menor o igual que m.
   - Si menor-igual(m) devuelve *verdadero*, significa que el número a adivinar está en el rango [c, m]. Por tanto, la función se llama recursivamente con estos nuevos límites: n := búsqueda-binaria2(c, m).
   - Si menor-igual(m) devuelve *falso*, el número a adivinar está en el rango [m+1, f], y la función se llama recursivamente con estos nuevos límites: n := búsqueda-binaria2(m+1, f).

Función `adivinar() dev n: nat`
Esta función SIMULA EL JUEGO COMPLETO, y no recibe parámetros. Su objetivo es encontrar el número a adivinar y para ello necesita establecer primero las cotas del rango dentro del cual se va a buscar.
1. **Establecimiento de Cotizaciones**: Inicialmente, cinf (cota inferior) se establece en 1 y csup (cota superior) también en 1. Luego, en un bucle, se duplica el valor de csup (csup := 2 * csup`) hasta que menor-igual(csup) devuelva verdadero, lo que indica que csup es ≥ que el número a adivinar. En cada iteración, cinf se actualiza al valor anterior de csup+1.
2. **Búsqueda Binaria**: Una vez establecidas las cotas, se llama a la función `búsqueda-binaria2` con cinf y csup como parámetros para encontrar el número a adivinar mediante el proceso de búsqueda binaria descrito anteriormente.

El algoritmo es eficiente porque en cada paso de la `búsqueda-binaria2`, el rango de búsqueda se reduce a la mitad, y el proceso para encontrar las cotas iniciales asegura que empezamos con un rango lo más ajustado

posible al número a adivinar. La función `menor-igual` es fundamental para este proceso, ya que proporciona la única información disponible para guiar la búsqueda: si el número a adivinar es menor o igual que el número propuesto.

11.2. Duende pasos

El pequeño duende Gumersindo quiere entrar en el País de las Hadas. Se encuentra delante de la frontera, formada por una enorme pared imposible de escalar que se extiende de forma infinita en ambas direcciones, y cuya única puerta es una abertura invisible que solo aparece cuando se pronuncian las palabras mágicas justo delante de ella.

Diseñar un algoritmo para que Gumersindo, que conoce las palabras mágicas pero no en qué dirección se encuentra la abertura, pueda encontrarla al cabo de un máximo de $O(n)$ pasos, siendo n el número de pasos que separan la posición actual de Gumersindo de la abertura.

--------------Solución-------------------

El problema de Gumersindo es que como no sabe en qué sentido (derecha o izquierda) se encuentra la abertura, no puede arriesgarse a caminar en una única dirección. Por eso debe combinar la búsqueda en una y en otra dirección (volviendo una y otra vez sobre sus pasos), a partir del punto inicial.

Supongamos que Gumersindo sigue la siguiente estrategia; empieza con un paso hacia la derecha (por ejemplo), pronuncia las palabras mágicas y, si no aparece la abertura, vuelve al punto de partida para dar un paso hacia la izquierda y probar suerte. Si no lo consigue, vuelve al punto inicial y repite

el proceso dando 2 pasos, *y así* sucesivamente.

Esta idea se ilustra en el gráfico de la Figura 11.1. donde se puede ver que en cada etapa se da un paso más que en la anterior y en la dirección contraria. Por tanto. Gumersindo conseguirá encontrar la abertura al cabo de $P(n) = \sum_{i=1}^{z(n)} i$ pasos, donde z(n)=2n-1 si comienza en la dirección adecuada y z(n)=2n si comienza en la dirección contraria con respecto a donde se encuentra la puerta. En cualquier caso, el número total *P(n)* de pasos que Gumersindo tiene que dar está en $\Theta(n^2)$.

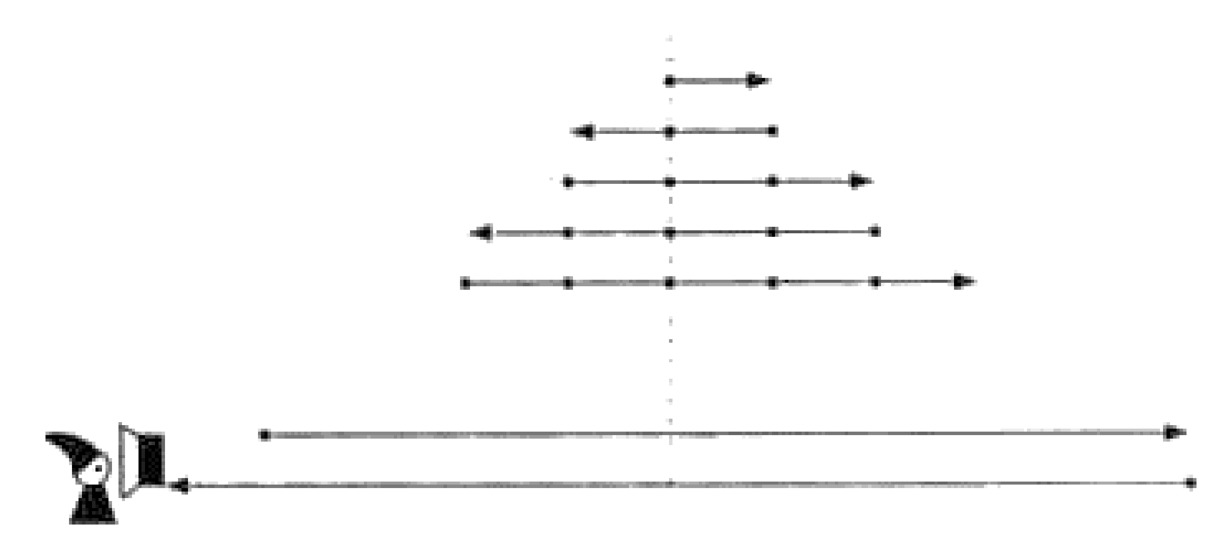


Figura 11.1: La búsqueda de Gumersindo.

¿Cómo conseguir la complejidad lineal pedida en el enunciado? Supongamos que Gumersindo, en lugar de incrementar de uno en uno los pasos que da en cada etapa, los duplica; es decir, comienza con 1 paso en una dirección, luego 2 en la otra, después 4. después 8. etc. (por supuesto, tendrá que pronunciar las palabras mágicas en cada posición nueva por la que pase). Con este nuevo algoritmo, el número total de

pasos que da Gumersindo hasta encontrar la abertura es menor que

$$\sum_{i=0}^{\lceil \log 2n \rceil + 1} 2^i = 2^{\lceil \log 2n \rceil + 2} - 1 \in \Theta\left(2^{\log 2n}\right) = \Theta(n)$$

11.3. Numero coincida con indice

Dado un vector ordenado *V* [1...*n*] de *n* números enteros distintos, escribir un algoritmo que en tiempo O (log n) encuentre un número *i* tal que 1<*i*<*n y* V[i]= *i.* siempre que exista.

----------------Solución-------------------

Para resolver el problema vamos a utilizar una idea similar a la búsqueda binaria (Ejercicio 11.1). Primero buscaremos en la posición central, *m,* del vector V[c..f]. En el caso de que *m*=V[m], habremos encontrado lo que buscábamos. Pero si *m*<V[m], entonces ni *m* ni las posiciones siguientes podrán cumplir la propiedad, pues por estar el vector ordenado y ser todos los elementos distintos tenemos que

m < V[m] < V[m+1] ⇒ m+1 < V[m+1].

Por tanto, buscaríamos en V[c..m-1]. Análogamente, si *m*>V[m], ni *m* ni las posiciones anteriores cumplirían la propiedad, por lo que buscaríamos en V[m+1..f].

El algoritmo que implementa estas ideas es el siguiente:

```
fun coincide?(V[1 ..n] de ent, c,f: nat) dev
⟨existe: bool, i: 1..n⟩
casos
    c>f → existe := falso
 [] c=f → ⟨existe, i⟩ := ⟨V[c] = c, c⟩
 [] c<f →
  m : = (c+f) div 2
  casos
      m<V[m] → ⟨existe,i⟩:=coincide?(V, c, m-1)
   □ m =V[m] → ⟨existe,i⟩:=⟨cierto, m⟩
```

```
□ m >V[m] → ⟨existe,i⟩:=coincide?(V, m+1,f)
fcasos
fcasos
ffun
```

El **COSTE** del algoritmo viene definido por la siguiente recurrencia:

$$T(n) = \begin{cases} c_0 & n \leq 1 \\ T(n/2) + c_1 & n > 1 \end{cases}$$

Por tanto, el coste del algoritmo está en Θ(log n), como se pedía.

CHATGPT

El pseudocódigo presentado aborda el problema de encontrar un índice i en un vector ordenado V[1..n] de números enteros distintos, tal que V[i]=i, utilizando un enfoque similar a la búsqueda binaria para lograr una complejidad de tiempo O(log n). Este enfoque es eficaz debido a la naturaleza ordenada y a la unicidad de los elementos del vector, lo que permite descartar mitades del vector en cada paso de la búsqueda.

Función coincide?(V[1..n] de ent, c, f: nat) dev ⟨*existe*: bool, i: 1..n⟩

La función coincide? busca en el subvector V[c..f] un índice i tal que V[i]=i. Devuelve un par ⟨*existe*, *i*⟩, donde *existe* es un booleano que indica si se ha encontrado tal índice, y i es el índice en cuestión (si existe).

1. **Caso Base 1** (c > f): Si la cota inferior c es mayor que la cota superior f, el subvector no tiene elementos y por lo tanto no se puede encontrar un índice que cumpla la condición. En este caso, *existe* se establece en falso.
2. **Caso Base 2** (c = f): Si c es igual a f, solo hay un elemento en el subvector. Se comprueba si ese único elemento cumple V[c]=c. Si es así, *existe* se establece en cierto y i en c; de lo contrario, *existe* se establece en falso.
3. ****Caso Recursivo** (c < f): Si hay más de un

elemento en el subvector, se calcula el índice medio `m` como (c+f)/2. Luego, se evalúan 3 subcasos basados en la comparación entre m y V[m]:

- Si m < V[m]: Dado que el vector está ordenado y todos los elementos son distintos, si m < V[m], entonces m+1 < V[m+1], m+2 < V[m+2], etc. Esto significa que ningún índice en el subvector V[m..f] puede cumplir V[i] = i. Por tanto, la búsqueda se continúa recursivamente en la primera mitad del subvector, V[c..m-1].
- Si m = V[m]: Se ha encontrado un índice m que cumple V[m] = m, por lo que existe se establece en cierto y i en m.
- Si m > V[m]: Análogamente al primer subcaso, si m > V[m], entonces m-1 > V[m-1], m-2 > V[m-2], etc. Esto indica que ningún índice en el subvector V[c..m] puede cumplir V[i] = i. La búsqueda se continúa en la segunda mitad del subvector, V[m+1..f].

El algoritmo termina cuando se encuentra un índice que cumple la condición V[i]=i o cuando se ha determinado que tal índice no existe dentro del rango especificado. Gracias a la estrategia de dividir el espacio de búsqueda a la mitad en cada paso, el algoritmo garantiza una complejidad de tiempo O(log n), lo cual es altamente eficiente para vectores de gran tamaño.

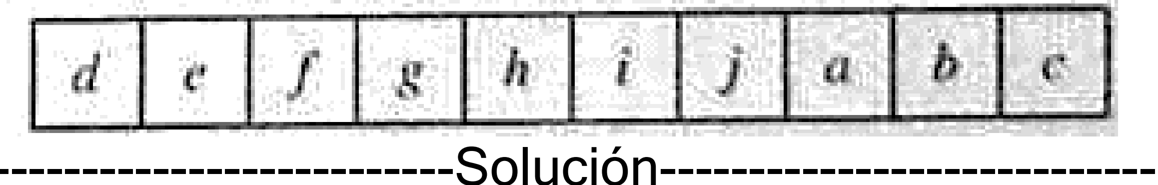
11.4. Trasponer vector

Dados un vector V[1..n] y un número natural *k* entre 1 y *n*-1,se pide diseñar un algoritmo eficiente que transponga los *k* primeros elementos de V con los elementos de las *n*-*k* últimas posiciones, sin hacer uso de un vector auxiliar.

Por ejemplo: si k=3 y si *V* es el siguiente vector de 10 elementos formado por las letras

(a b c d e f g h i j)

el resultado deseado sería (d e f g h i j a b c)



------------------------------Solución------------------------------

Si *k* = n-*k*, es decir, el número de elementos en los 2 subvectores que hay que transponer coincide, para resolver el problema basta realizar un recorrido desde 1 hasta *k* que vaya intercambiando 2 elementos cada vez. Como nos va a ser útil este algoritmo más tarde, lo detallamos a continuación, generalizando al dar como parámetros índices c_1, c_2, que indican el comienzo de los subvectores que se transponen, y *t* que indica el tamaño común de ambos. El número de intercambios de elementos que se hace es igual a *t*.

$\{1 \leq c_1 < c_2 \leq n \wedge c_1 + t \leq n+1 \wedge c_2 + t \leq n+1\}$

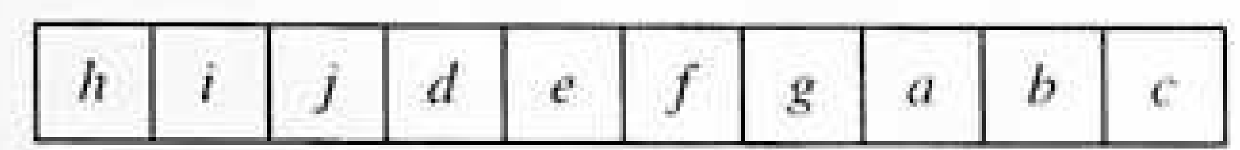
proc **transponer-iguales**(V[1..n] de

elemento, **e** c_1*,*c_2*,t :*1..*n)*

i := 0
mientras *i* < *t* **hacer**
⟨V[c_1+i, V[c_2+i]⟩ := ⟨V[c_2+i], V[c_1+i]⟩
i := *i* + 1
fmientras
fproc

Supongamos ahora que $k \neq n\text{-}k$ y consideremos el subvector más corto. Tomando el ejemplo del enunciado, comenzamos intercambiando los 3 primeros elementos del vector con los 3 últimos:

(h i j d e f g a b c)



Ahora ya tenemos bien colocados los k=3 últimos y solo tenemos que seguir trabajando con los primeros n-k=7 elementos. En concreto, debemos transponer los k primeros con los 7-k últimos; es decir, hemos reducido el problema original a uno similar, pero de menor tamaño. En general, cuando $k < n\text{-}k$. la situación será que tras este primer paso que se realiza usando el procedimiento "transponer-iguales" quedan colocados en su lugar definitivo los k últimos elementos (que vienen del principio del vector) y hay que continuar transponiendo los k primeros (que vienen del final del vector) con los n-$2k$ siguientes, lo cual se puede hacer recursivamente. La situación se muestra

en el gráfico de la Figura 11.2(a) donde se desea transponer un subvector *I* a la izquierda con uno *D* mayor a la derecha que se descompone en *D1* y D2 para intercambiar dos subvectores de la misma longitud *I* y D_2. La llamada recursiva tendrá que hacerse para transponer D_2 con D_1, y así colocar bien todos los elementos de *D.* Los índices que aparecen en la figura corresponden a los usados en el algoritmo recursivo que veremos más adelante.

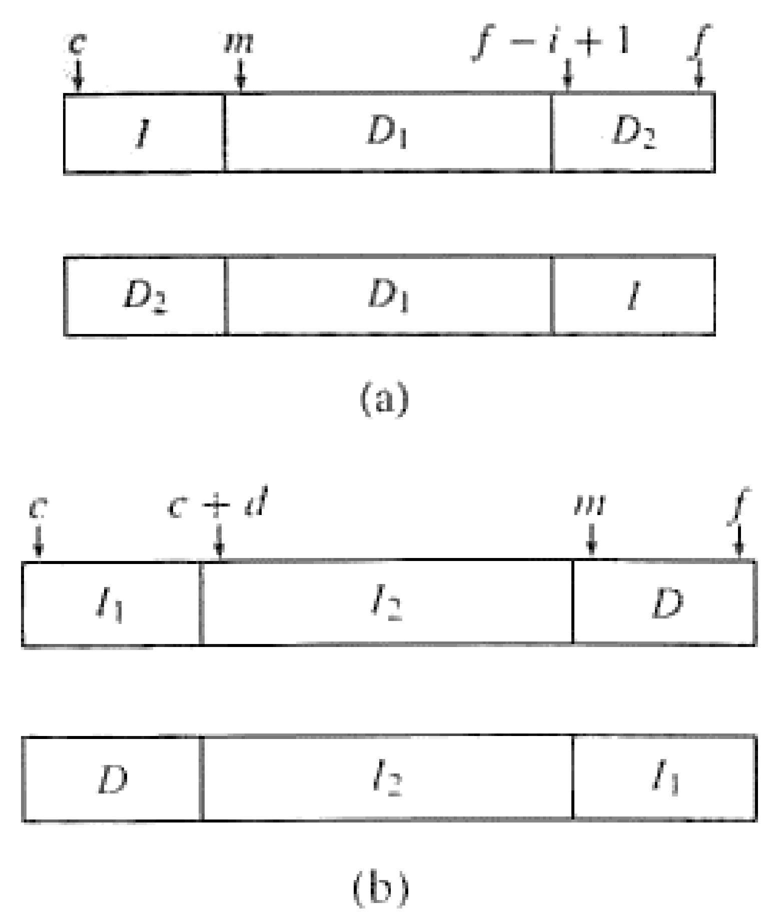


Figura 11.2: Esquemas para la transposición de 2 subvectores.

Cuando k >*n-k* no podemos intercambiar los k primeros con los n-*k* últimos, porque los elementos solapan. Ello nos obliga a considerar el subvector más corto para reducir el número de elementos a intercambiar, en concreto, intercambiaremos los *n-k* primeros con los *n-k* últimos, siendo ahora los *n-k*

primeros los que quedarán definitivamente colocados y resolviendo recursivamente el problema con los restantes. Esta situación simétrica a la anterior se muestra en el gráfico de la Figura 11.2(b), donde ahora hace falta descomponer el subvector de la izquierda.

Como las llamadas recursivas se hacen con subvectores del vector original, usamos índices *c* y *f* para indicar los extremos del subvector en que se trabaja en cada llamada recursiva. El parámetro *m* indica la posición donde comienza el segundo subvector de los dos que se transponen. La llamada inicial para resolver el problema del enunciado se hará con $c=1$, $m=k+1$ y $f=n$.

El algoritmo resultante queda como sigue, donde la única dificultad es tener un poco de cuidado al calcular los índices para las llamadas recursivas (los diagramas de la Figura 11.2 ayudan a su comprensión). Específicamente, *i* denota la longitud del subvector de la izquierda y *d* la del subvector de la derecha.

$(1 \leq c \leq m \leq f \leq n)$

```
proc transponer(V[1..n] de elemento, e c,m,f : 1..n)
i:= m-c ;
d:= f -m+1
casos
   i=d → transponer-iguales(V, c,m, i)
 □ i<d → transponer-iguales(V, c, f-i+1, i)
         transponer(V, c, m, f-i)
 □ i> d →transponer-iguales(V, c, m, d)
```

```
        transponer(V, c+d, m, f)
    fcasos
fproc
```

Como este algoritmo recursivo es lineal y final, se obtiene inmediatamente una versión iterativa que ya escribimos para la situación inicial del enunciado.

{1≤k<n}

```
proc transponer-iter(V[1..n] de elemento,
e k : nat)
 c:=1;
 m:= k+1;
 f:= n;
 i:= m-c;
 d:= f-m+1
 mientras i ≠ d hacer
   si i < d entonces
     transponer-iguales(V, c, f-i+1, i);
     f := f-i;
     d:= d-i
   sino
     transponer-iguales(V, c, m, d) ;
     c:= c+d;
     i:= i-d
   fsi
 fmientras
 transponer-iguales(V, c, m, i)
fproc
```

El número de intercambios de elementos realizados por transponer depende del tamaño de los dos subvectores que se transponen y que llamamos i y *d* como en el cuerpo del algoritmo, dando lugar a la siguiente recurrencia:

$$I(i,d) = \begin{cases} i & i = d \\ i + I(i, d-i) & i < d \\ d + I(i-d, d) & i > d \end{cases}$$

Excepto por las sumas que se hacen en los casos recursivos, se puede reconocer la semejanza que guarda esta recurrencia con la versión recursiva del *algoritmo de Euclides* para calcular el máximo común divisor, y se puede comprobar que su solución exacta es I(i,d) = *i*+*d* - mcd(i,d). Por tanto, el coste en tiempo de `transponer-iter` y de `transponer` es lineal con respecto a la longitud del vector, mientras que el coste en espacio es constante pues, como se requería en el enunciado, la transposición se realiza sin ayuda de ningún vector auxiliar.

CHATGPT

El pseudocódigo presentado describe un algoritmo para transponer elementos dentro de un vector V[1..n] sin utilizar un vector auxiliar. La transposición implica intercambiar los k primeros elementos del vector con los n-k últimos elementos. Este proceso se realiza de manera eficiente mediante el uso de la recursión o iteración, dependiendo de la relación entre k y n-k.

Procedimiento transponer-iguales(V[1..n] de elemento, c1,c2, t: 1..n)

Este procedimiento se encarga de intercambiar dos subvectores de igual tamaño t dentro del vector V, donde los subvectores comienzan en las posiciones c1 y c2, respectivamente. Realiza t intercambios, moviendo el elemento en la posición c1+i al lugar del elemento en la posición c2+i, y viceversa, para cada i desde 0 hasta t-1.

Procedimiento transponer(V[1..n] de elemento, c, m, f: 1..n)

Este procedimiento es el núcleo del algoritmo y maneja la transposición de subvectores de diferentes tamaños.

Los parámetros c, m, y f representan, respectivamente, el inicio del subvector izquierdo, el inicio del subvector derecho (después del subvector izquierdo), y el final del subvector derecho.

1. **Caso i = d:** Si los subvectores izquierdo y derecho tienen el mismo tamaño i=m-c y d=f-m+1, se utiliza `transponer-iguales` para intercambiarlos directamente.
2. **Caso i<d**: Si el subvector izquierdo es más corto, se transponen los i elementos del subvector izquierdo con los últimos i elementos del subvector derecho usando `transponer-iguales`. Luego, se hace una llamada recursiva a `transponer` para manejar el subvector restante, reduciendo el problema a un tamaño menor.
3. **Caso i>d**: Si el subvector izquierdo es más largo, se transponen los d elementos del subvector derecho con los primeros d elementos del subvector izquierdo. Posteriormente, se realiza una llamada recursiva a `transponer` para ajustar los elementos restantes en el subvector izquierdo.

Procedimiento transponer-iter(V[1..n] de elemento, k: nat)

Este procedimiento implementa la versión iterativa del algoritmo, manejando el caso inicial en el que se desea transponer los k primeros elementos con los n-k últimos. A través de un bucle, ajusta los índices c, m, y f para las llamadas a `transponer-iguales`, dependiendo de si el subvector izquierdo (i = m-c) es más corto o más largo que el subvector derecho (d = f-m+1). El bucle continúa hasta que los subvectores izquierdo y derecho tienen el mismo tamaño, momento en el cual se realiza un último intercambio mediante `transponer-iguales`.

Este enfoque aprovecha la eficiencia de la recursión (o iteración en la versión iterativa) para reducir progresivamente el problema a subproblemas más pequeños, permitiendo intercambiar elementos sin necesidad de un espacio auxiliar adicional, y manteniendo una complejidad temporal óptima. La clave del algoritmo radica en su capacidad para adaptarse a diferentes relaciones entre k y n-k, manejando tanto casos simétricos como asimétricos de manera eficiente.

11.5. Mergesort

Desarrollar una versión iterativa "ascendente" de la *ordenación por mezclas* (**mergesort**) consistente en realizar los siguientes pasos: se comienza considerando parejas de elementos consecutivas, las cuales se ordenan; a continuación se mezclan, de dos en dos, las parejas ordenadas: después se mezclan los cuartetos ordenados: y así sucesivamente, hasta obtener el vector completo ordenado.

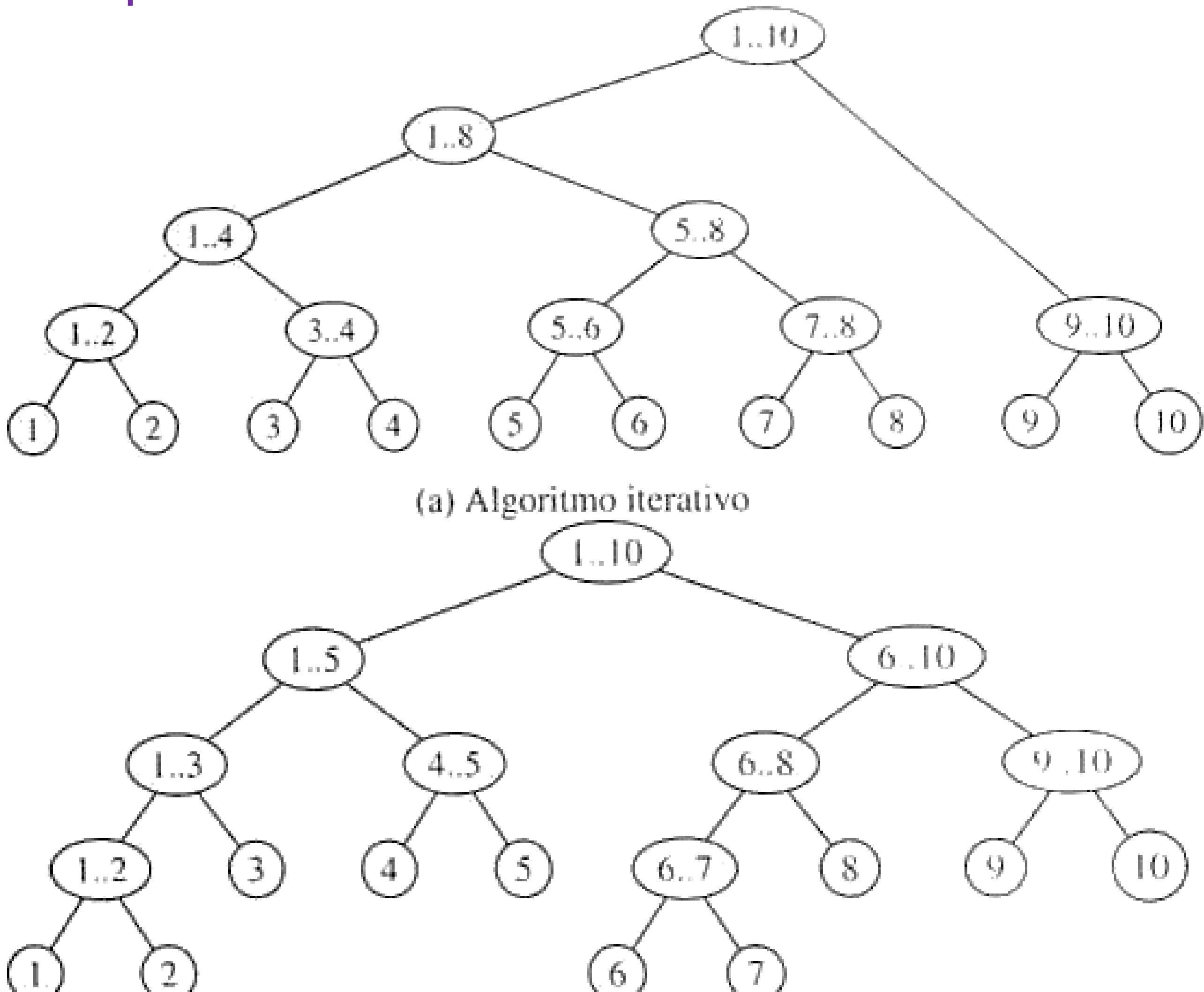


Figura 11.3: Árboles de etapas para la ordenación por mezclas.

-------------------------------Solución--------------------------

Como indica el enunciado, el algoritmo consiste en una serie de etapas de mezclas de pares de subvectores (consecutivos) ordenados, de tamaño

uniforme. Se comienza con subvectores unitarios, duplicando el tamaño en cada etapa, hasta alcanzar el tamaño del vector completo. En la Figura 1 1.3 se muestran las distintas etapas para un vector de 10 elementos; el primer árbol corresponde al algoritmo iterativo aquí propuesto, y el segundo árbol corresponde al algoritmo recursivo "clásico".

Para realizar las mezclas es necesario utilizar un vector auxiliar, teniendo que copiar los datos desde dicho vector auxiliar hacia el original al final de cada etapa. Podemos reducir el número de copias organizando las etapas por pares, de forma que primero se mezcla en el auxiliar, en la siguiente etapa se mezcla en el original, y así sucesivamente se van alternando.

El algoritmo que recoge estas ideas es el siguiente:

```
proc mergesort-iter(V[1..n] de elemento)
var W[1..n] de elemento        {vector auxiliar}

 tamaño := 1          {tamaño de cada subvector}
 mientras tamaño < n hacer
    etapa-mezclas(V, W, tamaño)
    tamaño := 2 * tamaño
    etapa-mezclas(W, V, tamaño)
    tamaño := 2* tamaño
 fmientras
fproc
```

En cada etapa se recorre todo el vector y se realizan todas las mezclas posibles. En cada caso, para realizar una mezcla se

necesitan *2*tamaño* elementos. Cuando ya no hay suficientes elementos para realizar una mezcla "completa", si aún quedan más de *tamaño,* se formará un subvector de *tamaño* elementos que se mezclará con los elementos restantes. Pero si solo quedan *tamaño* o menos, dichos elementos sobrantes simplemente se copian en el vector de destino.

```
proc etapa-mezclas(e V[1..n] de elemento,
W[1..n] de elemento, e tamaño: nat)
i := 1        {posición actual en el vector}
mientras i ≤ (n - 2 * tamaño + 1) hacer
    mezclar(V, W, i, i+tamaño-1, i+2*tamaño-1);
    i := i+2*tamaño
fmientras

{quedan menos de 2 * tamaño elementos}
si n-i+1 > tamaño entonces {quedan más de
tamaño elementos}
    mezclar(V, W, i, i+tamaño-1, n)
si no        {no hay suficientes para mezclar}
    W[i..n] := V[i..n]
fsi
fproc
```

Para completar la solución, damos a continuación el algoritmo que efectúa la mezcla de 2 subvectores ordenados contiguos, que coincide con el que se utiliza en la versión recursiva clásica de la ordenación por mezclas.

{1≤c≤m<f ∧ V[c] ≤... ≤V[m] ∧ V[m+1]≤...≤ V[f]}

```
proc mezclarte V[1..n] de elemento. W[1..n] de
elemento, e c,m,f: nat)
 i := c;
 j := m+1;
 k := c
```

```
mientras i≤m ∧ j≤f hacer
    si V[i]≤V[j] entonces
        W[k]:= V[i];
        i:= i+1
    si no
        W[k]:= V[j];
        j:= j+1
    fsi
    k := k+1
fmientras

si i>m entonces   {copiar elementos del segundo subvector}
    W[k..f] := V[j..f]
si no    {j>f, copiar elementos del primer subvector}
    W[k..f] := V[i..m]
fsi
fproc
```

{W[c] ≤…≤ W[f]}

El coste en tiempo de una etapa de mezclas es lineal respecto al número total de elementos en el vector y se realizan logn etapas, por lo que el coste en tiempo de mergesort-iter está en Θ(n logn), como el coste de la versión recursiva clásica. El presente algoritmo tiene la ventaja adicional de que, al ser iterativo, el espacio auxiliar se reduce a la variable local W.

11.6. Ordenación con indices

En algunas aplicaciones se necesita conocer la ordenación de los elementos de un vector, pero se desea mantener el vector sin modificar. En esos casos, es conveniente un algoritmo de ordenación que devuelva un vector de *índices* tal que la componente í-ésima indique la posición (en el vector de entrada) del elemento que debe ocupar el *i*-ésimo lugar en la ordenación deseada.

Hacer una versión del algoritmo de *ordenación por mezclas* que obtenga este vector de índices.

------------------------Solución-------------------------

Vamos a seguir el esquema recursivo de la ordenación por mezclas. Las modificaciones son muy pocas y afectan solamente a la consulta de los valores del vector a ordenar, que ahora hay que realizar de forma "indirecta" a través del vector de índices.

```
proc mergesort-indices(e V[1..n] de
elemento, índices[1..n] de 1..n. e c, f : nat)
casos
      c > f → nada
    ▯ c = f→ índíces[c] := c
    ▯ c < f →
        m := (c + f) div 2
        mergesort-índices(V, índices, c, m)
        mergesort-índicesí V, índices, m+1, f)
        mezclar-índices(V, índices, c, m, f)
fcasos
fproc
```

Así mismo, el algoritmo mezclar-índices

sigue la misma estructura que el algoritmo mezclar del Ejercicio 11.5.

{ 1 ≤ *c* ≤ *m* < *f* ≤*n* ∧ V[I[c]] < ... < V[I[m]] ∧ V[I[m+1]] < ... < V[I[f]]}

```
proc mezclar-índices(e V(1 ..n] de elemento, I[1..n] de 1..n, e c, m, f : 1..n)
var aux[1..n] de 1..n
i := c ; j := m+1 : k := c
mientras i ≤ m ∧ j ≤ f hacer
   si V[I[i]] ≤ V[I[j]] entonces
      aux[k] := I[i] ; i := i + 1
   si no
   aux[k] := I[j] ; j := j+1
fsi
  k := k+1
fmientras
si i > m entonces {copiar elementos del segundo subvector}
   aux[k..f] := I[j..f]
si no {j > f, copiar elementos del primer subvector}
    aux[k..f] := I[i..m]
fsi
I[c..f] := aux[c..f]
fproc
  { V[I[c]]≤...≤ V[I[f]]}
```

El algoritmo solicitado es la siguiente función, que hace una llamada inicial a mergesort-índices.

```
fun ordenar-índices(V[1..n] de elemento)
dev indices[1..n] de 1..n
          mergesort-índices(V. índices, 1, n)
ffun
```

El coste en tiempo es del mismo orden que el de la ordenación por mezclas usual, es decir, que está en $\Theta(n \log n)$.

11.7. Ordenación

Durante el proceso de ***ordenación por mezclas*** los elementos se copian, en general, múltiples veces de una posición a otra en el vector.

Desarrollar una versión eficiente en la que cada elemento se copie una única vez, desde su posición inicial hacia la posición final en el vector ordenado, utilizando una estructura auxiliar que refleje el orden implícito de los elementos.

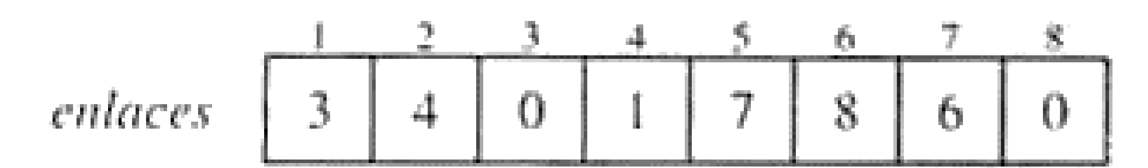


Figura 11.4: Vector de enlaces con dos listas.

---------------------------Solución----------------------

La idea consiste en representar cada elemento por su posición en el vector inicial. Entonces, los subvectores ya ordenados se representan mediante listas de posiciones. La implementación de estas listas se puede hacer sobre un vector de *enlaces* del mismo tamaño, de forma que cada entrada en el vector de enlaces indica la posición del siguiente elemento, y el valor 0 indica el final de una lista. Las mezclas se realizan sobre el vector de enlaces, que es el que se modifica, mientras que V se mantiene inmutable hasta el final del algoritmo, en que se utilizará el vector de enlaces final para obtener la ordenación de V.

Por ejemplo, el vector de enlaces de la Figura 11.4 contiene 2 listas: el número 2 denota el comienzo de la lista *l1* = (2,4,1,3), y el número 5 el comienzo de la lista *l2* = (5,7,6,8). Si interpretamos las listas como que están describiendo el orden en el vector V[1..8] la conclusión es que V[2]≤V[4] ≤V[1]≤V[3] y V[5]≤V[7] ≤V[6]≤V[8].

El algoritmo queda así:

proc mergesort-enlaces(**e** V[1..n] **de** *elemento, enlaces[0..n]* **de** 0*..n,* **e** c,*f:nat, p*: **O..n)**

{p será la posición inicial de la lista correspondiente a V[c..f]}

casos

c > f → **nada**

 c = f → *p := c*

 c<f →

m:= (c+f) div 2

mergesort-enlaces(*V,enlaces,c,m, p1)*

mergesort-enlaces(V,*enlaces,m*+1, *f, p2)*

mezclar-enlaces(V, *enlaces, p1, p2,p)*

fcasos

fproc

Con la llamada inicial:

enlaces[0..*n*] := [0]

mergesort-enlaces(*V,enlaces,* 1,*n,p)*

ordenar-enlaces(V, *enlaces, p)*

Nótese que el vector *enlaces* se ha definido con *n*+1 posiciones: la posición adicional *enlaces*[0| simplifica el algoritmo para mezclar dos listas ordenadas de enlaces, que damos a continuación:

{p1 es el comienzo de la primera lista, *p2* es

el de la segunda y *p* será el de la lista mezclada}

proc mezclar-enlaces(**e** V[1..n] de *elemento. enlaces[0..n]* **de** *0,.n. e p1, p2* : 0..n, *p*: 0..n)

i:= p1 : j:= p2 : k := 0

{*k* es el último en la lista que se construye}

mientras *i* ≠ 0 ∧ *j* ≠ 0 **hacer**

si V[i] ≤ V[j] **entonces**

enlaces[k] := i ; k := i ; i := enlaces[i]

si no

enlaces[k] := j ; k := j ; j := enlaces[j]

fsi

fmientras

			0	1	2	3	4	5	6	7	8	
		V	-	24	12	80	7	15	43	29	20	
		enlaces	0	0	0	0	0	0	0	0	0	
p_1	p_2	p										
1	2	2	2	0	1	0	0	0	0	0	0	(12, 24)
3	4	4	4	0	1	0	3	0	0	0	0	(12, 24), (7, 80)
2	4	4	4	3	1	0	2	0	0	0	0	(7, 12, 24, 80)
5	6	5	5	3	1	0	2	6	0	0	0	(7, 12, 24, 80), (15, 43)
7	8	8	8	3	1	0	2	6	0	0	7	(7, 12, 24, 80), (15, 43), (20, 29)
5	8	5	5	3	1	0	2	8	0	6	7	(7, 12, 24, 80), (15, 20, 29, 43)
4	5	4	4	7	5	0	2	8	3	6	1	(7, 12, 15, 20, 24, 29, 43, 80)

Tabla 11.1: Modificaciones sobre *enlaces* al ordenar un vector *V*.

si *i* = O **entonces** enlaces[k] := *j* { enlazar resto del segundo vector)

sino enlaces[k] := *i* { enlazar resto del primer vector}

fsi

p := enlaces[0] {comienzo lista construida}

fproc

Obsérvese cómo gracias a *enlaces*[0] siempre existe el "último" en la lista que se va construyendo, incluso cuando esta está totalmente vacía. Nótese que este algoritmo tiene exactamente la misma estructura que el algoritmo mezclar del Ejercicio 11.5 pero en vez de utilizar contadores explícitos se pasa de un

elemento al siguiente mediante el vector *enlaces* lo cual elimina también la necesidad de hacer copias.

Finalmente, el algoritmo para obtener el vector ordenado es el siguiente:

proc ordenar-enlaces(V[1..n] **de** *elemento,* **e** enlaces[0.,n] **de** 0..n, **e** *p :* 0..n)
var W[1..n] **de** *elemento*
i := p
para *k* = **1 hasta** *n* **hacer**
W[k] := V[i] ; *i*:=enlaces[í]
fpara
V := W
fproc

Para comprender mejor el algoritmo, veamos cómo funciona para ordenar el siguiente vector:

	1	2	3	4	5	6	7	8
V	24	12	80	7	15	43	29	20

La Tabla 11.1 muestra cómo se actualiza el vector *enlaces* cada vez que termina una llamada a mergesort-enlaces. En cada fila, *p1* y *p2* indican los valores devueltos por las llamadas recursivas a mergesort-enlaces y *p* el comienzo de la lista producida por mezcla-enlaces. A la derecha se muestran los subvectores ordenados que representan estas listas. Por ejemplo, en la última lila se tiene *p*=4, indicando dónde empieza la lista de índices 4,2,5,8,1,7,6,3, que define el siguiente orden en el vector:

V[4]≤V[2]≤V[5] ≤V[8]≤ V[1]≤V[7]≤ V[6]≤V[3]

11. Divide y venceras

11.8. Tornillo (demostr)

En un habitación oscura se tienen 2 cajones, en uno de los cuales hay n tomillos de varios tamaños, y en el otro las correspondientes *n* tuercas. Es necesario emparejar cada tornillo con su tuerca correspondiente, pero debido a la oscuridad no se pueden comparar tornillos con tornillos, ni tuercas con tuercas, y la única comparación posible es la de intentar enroscar una tuerca en un tornillo para comprobar si es demasiado grande, demasiado pequeña, o se ajusta perfectamente al tomillo.

a) Demostrar que cualquier algoritmo para resolver este problema debe realizar $\Omega(n \log n)$ comparaciones en el caso peor.

b) Desarrollar un algoritmo para emparejar los tornillos con las tuercas que use $O(n \log n)$ comparaciones en término medio.

------------------------Solución:------------------------

Apartado (a)

Para demostrar que cualquier algoritmo para resolver este problema debe realizar Q $(n \log n)$ comparaciones en el caso peor, hay que estudiar el *árbol de decisión* correspondiente a un algoritmo correcto cualquiera. Si consideramos que una comparación de una tuerca con un tornillo tiene 3 posibles resultados (menor, encaja, mayor), entonces el árbol de decisión es ternario (cada nodo tiene 3 hijos).

Ahora bien, existen *n*' formas de emparejar *n* tornillos con *n* tuercas, por lo que se necesitan al menos *n*! hojas en el árbol de decisión.

Un árbol ternario de altura *h* tiene un máximo de 3^{h-1} hojas (como se puede demostrar de forma parecida al Ejercicio 8.3). y en nuestro árbol de decisión cada nivel menos el último corresponde a realizar una comparación. Por tanto, si *T(n)* representa el número de comparaciones realizadas por el hipotético algoritmo para *n* tuercas (y *n* tornillos), entonces se tiene 3'> *n*'.. es decir. *Tinl* > log^/il). Ahora bien, ¿cómo de grande es logjOi!)? Al respecto, tenemos que

n

//'- = (1 * 2 * ... * >2)(//*... * 2 * 1) = j"[*k(n* + 1 — Ár).

;=i

Vamos a acotar *nl~* inferior y superiormente considerando el menor valor *(k* = 1 y *k* = *n)* y el mayor valor *(k* = *(n* + 1)/2) posibles:

Á = I A = 1 x " 7 x

por tanto,

y tomando logaritmos.

// / *n* + 1 \

- log; *n* < log; (>2!) < *n* log; I

—-— I .

Así pues. log3 (/2!) e O(n log; //), con lo que concluimos que *T(n)* € logn).

Apartado (b)-------------------------------
Para resolver el problema, utilizamos una idea semejante a la de la *ordenación rápida* (quicksort). Cuando el número de tuercas es n < 1 estamos en el caso inmediato: el emparejamiento es trivial y no hay que hacer nada. Pero cuando n > 1. procedemos de la siguiente manera: se coge un tornillo cualquiera y. comparando con ese tornillo, se divide el conjunto de tuercas en tuercas menores, tuercas mayores y tuercas que enroscan. A continuación, usando una de las tuercas que enroscan, se divide el conjunto de tomillos en menores, mayores e iguales. Basta ahora hacer sendas llamadas recursivas con los conjuntos de tuercas y tomillos menores y mayores.

Si el algoritmo se implementa usando vectores, la división corresponde a una partición con pivote dado. El algoritmo que típicamente se utiliza para realizar la partición en la ordenación rápida devuelve el vector transformado:

<	*pivote*	>

Pero esta forma de partición solo es correcta en el caso de que todos los tornillos (y tuercas) sean de tamaños diferentes, porque si no, al hacer las dos particiones podrían no coincidir las posiciones de los pivotes, con lo que en las llamadas recursivas los conjuntos de tuercas y tornillos serían de tamaños distintos. Se hace necesaria una partición que nos devuelva el vector dividido en

tres partes:

El algoritmo para emparejar las tuercas con los tornillos es el siguiente (donde cada tornillo y tuerca se representan por su tamaño):

proc emparejar(rorm7/os[l.j7], ruercosjl.jt] de nat4-, **e** *c, f : nat)*
  si c < *f* entonces
    partición *{tuercas, c. f. tornillos[c], i. j)*
    partición(romíZ/or, *c, f, tuercas[i], k, l)*
    {i =k A j=l}
    emparejar(ronu7/oj, *tuercas, c, i* — 1)
    emparejar(romí//os, *tuercas, j* + 1, *f)*
  fsi
fproc

Para hacer la partición del vector utilizaremos tres índices *i, j y k.* Los índices *i* y *k* se moverán hacia la derecha mientras que el índice *J* se moverá hacia la izquierda, de forma que se tiene es decir, que los elementos en V[c..í — 1] son menores que *pivote,* los elementos en V|(..A — 1] son iguales *a pivote,* los elementos en V[A'..j] están todavía sin colocar y se desconoce su relación con *pivote,* y los elementos en V[j + 1../] son mayores que *pivote.* En cada paso, el algoritmo compara V[A'] con el pivote:

(a) Si V[¿] es menor, se intercambia con V[í] para colocar el elemento menor junto con los demás menores. Al realizar el intercambio, se coloca en V [A] un elemento igual a *pivote,* por lo que, en este caso, se incrementan los índices *i* y *k.*

(b) Si V[A-] es igual a *pivote,* entonces está bien colocado, por lo que se incrementa *k.*

(c) Si V[Z.'] es mayor que *pivote,* se intercambia con V[J] para colocarlo junto con los elementos mayores. En este caso el elemento colocado en L[/.'] después del intercambio es "desconocido”, por lo que solo podemos avanzar (decrementándolo) el índice *j.*

El bucle termina cuando *k* > *j,* lo que indica que no quedan elementos mal colocados. El algoritmo es el siguiente:

proc partición(V] I ,.n] **de** *elemento, e c. f : nat.* **e** *pivote : elemento, i. j : nat)*

{ los índices *i y j* son parámetros de salida y describirán la partición obtenida}

i := c ; k := c ; j := f

mientras *k* < *j* **hacer casos**

V[<) < *pivote* ->(V[*], V[/J) := (Vi<]. V[JtJ} :*i := i* + 1 ;*k := k* + 1

▫ VIA’I = *pivote* *k := k* + 1

▫ V|Á] > *pivote* ->{ V|¿J. V[y]> := (V[7J. V[Jt]) ; *j := j* -1

fcasos

(mientras fproc

((V/- : *c* < *r* < *i* : V[r] < *pivote)* A (Vs *: i* < *s* < *j* : V[s) = *pivote)*

A (V/ : *j* < *t* < *f* : V(r| > *pivote))*

Como se puede apreciar, partición realiza del orden de *n* comparaciones. Por tanto, cuando hay *n* tuercas y *n* tornillos, el número de comparaciones del algoritmo emparejar viene determinado por una recurrencia de la forma:

T. _ I cu *n* < 1

" | *T{p) + T(q) + C\n n*-> I
siendo *p* y *q* los tamaños respectivos de los conjuntos de tuercas (y tornillos) menores y mayores, y cumpliéndose *p* + *q* < *n.*

Una demostración análoga a la vista en la solución del Ejercicio 6.6 prueba que *Tin)* € O(/i") independientemente de la elección de *p y q.*

Como ocurre en la ordenación rápida, el caso peor se presenta cuando, siendo todos los tomillos (y tuercas) de diferentes tamaños, la división es desequilibrada y el pivote se coloca sistemáticamente en uno de los extremos (tornillo/tuerca más grande o pequeño del conjunto tratado), dando lugar en este caso a una recurrencia de la forma:

í cg *n* < 1
{ T_p(*n* - l)+co + ci„ «> 1

Así pues. *T,,(n)* e 0(n^2).

El coste en tiempo en el caso medio está determinado por la variación de los tamaños de *p* y </. El análisis es similar al de la ordenación rápida, resultando *T„,(n) e &(n* logn). cuya demostración se puede ver en [BB97. Sección 7.4.2].

11.9. Seleccionar f esimo

Dado un vector V[1..n] de elementos que se pueden ordenar, se desea *seleccionar* el k-ésimo menor elemento del vector, o sea, el elemento que ocuparía la posición *k* del vector V[1..n] si este estuviera ordenado, para cualquier *k* entre 1 y *n.* Se ha de lograr, en el caso peor, un coste lineal con respecto al tamaño del vector.

----------------------------Solución-------------------------

En la primera solución que presentamos se hace una llamada al *algoritmo de partición* dado en la solución al Ejercicio 11.8, tomando como pivote el primer elemento del subvector:

Partición(V, c, *f,* V[c], i. *j)*

Si el índice *k,* correspondiente a la posición del elemento buscado, cae en la zona de los elementos que son iguales al pivote, o sea. $i \leq k \leq j$, ya hemos encontrado el elemento buscado; si no es así, se hace una llamada recursiva con el subvector correspondiente.

{*k* representa una posición absoluta, *c* < *k* < *f*}

```
proc selección1(V[1..n] de elemento, e c,f,k: nat, k-ésimo: elemento}
  si c = f entonces
     k-ésimo:= V[c]
  si no
     partición(V, c, f, V[c], i, j)
     casos
         k<i → selección1 (V, c.i-1, k, k-ésimo)
       ▯ i≤k ∧ k≤ j → k-ésimo := V[k]
       ▯ k>j → selección1 (V, j+1, f. k, k-ésimo)
```

fcasos
fsi
fproc

Nótese que en esta solución *k* representa una posición *absoluta,* referida en todo momento al vector inicial de rango 1..n; por eso *k* siempre se mantiene entre los límites del subvector donde se está buscando. Pero se pueden obtener soluciones donde *k* represente una posición *relativa,* de forma que *k* sea modificado adecuadamente cuando cambia el comienzo del subvector. Por ejemplo, si se desea encontrar el “tercer" elemento de un vector V[1..10], se comenzará con *k* = 3. pero si la búsqueda nos lleva al subvector V[3..7], deberemos entonces localizar el "primero" *(k* = 1).

El caso peor del algoritmo presentado tiene lugar cuando el pivote queda siempre en un extremo del subvector correspondiente, por lo que una llamada sobre un vector de tamaño *n* da lugar a una llamada recursiva sobre un subvector de tamaño *n* – 1, y en definitiva a un coste en el caso peor en $\Theta(n^2)$. Sin embargo, el coste en el caso medio es lineal con respecto al tamaño *n* del vector |HSR98. Sección 3.6]

La forma de mejorar este algoritmo es asegurarnos de que el pivote elegido no va a quedar en un extremo. La mejor elección sería tomar como pivote la *mediana* del subvector, pero calcular la mediana es precisamente un caso

particular del problema de selección, cuando $k = (c + f)$ div 2. En esta situación, nos conformaremos con una aproximación suficientemente buena de la mediana, conocida como *mediana de las medianas.* La idea consiste en dividir el vector original en trozos de 5 elementos, para cada uno de los cuales la mediana se calcula de forma directa. A continuación se calcula la mediana de esas *n* div 5 medianas mediante el algoritmo de selección. La elección de la mediana de las medianas como pivote hace que el caso peor que antes vimos ya no se pueda dar.

Para no utilizar espacio adicional, en el algoritmo que sigue se trasladan las medianas al principio del subvector sobre el que se trabaja. Además, en vez de considerar como caso básico el vector unitario, se consideran como casos básicos, que se resuelven directamente utilizando algún algoritmo de ordenación sencillo de buen comportamiento para pocos elementos (por ejemplo, ordenación por *inserción).* los vectores con menos de 13 elementos. La razón de considerar este valor concreto se puede encontrar en el libro de Brassard y Bratley [BB97, Sección 7.5] donde se demuestra que el coste en tiempo de este algoritmo, incluso en el caso peor, es lineal con respecto al tamaño *n* del vector, a pesar de la llamada recursiva adicional para el cálculo de la mediana de

las medianas.

{ c < k < f }

```
proc selección2(V[1..n] de elemento, e c,f,k:
nat, k-ésimo: elemento)
 t:= f-c+1
 si t ≤12 entonces
   ordenar-inserción (V,c,f)
   k-ésimo := V[k]
 si no
   s:= t div 5
   para l=1 hasta s hacer
     ordenar-inserción(V,c+5 *(l-1), c+5*l-1);
     pm := c+5*(l-1) + 5 div 2
     ⟨V[c+l-1], V[pm]⟩ := ⟨V[pm], V[c+l-1]⟩
   fpara
   selección2(V, c, c+s-1, c+(s-1) div 2, mm);
   {mm es la mediana de las medianas}
   partición(V, c, f, mm, i, j);
   casos
       k<i → selección2(V, c, i-1, k, k-ésimo)
     □ i≤k ∧ k≤j → k-ésimo := mm
     □ k>j → selección2(V, j+1, f, k, k-ésimo)
   fcasos
 fsi
fproc
```

En este algoritmo algunas de las expresiones de los índices se pueden simplificar, pero resulta más ilustrativo no hacerlo. Por ejemplo, *c+5*(l-1)*+5 div 2 indica la posición de la mediana del l-ésimo trozo de 5 elementos, mientras que *c*+(s-1) div 2 es la posición de la mediana de las medianas después de haber colocado las s medianas en el subvector V[c..c+s-1].

11.10. 3 probl equivalentes

Dado un vector V[1..n] de *n* elementos que se pueden ordenar, demostrar que los 3 problemas siguientes son *linealmente equivalentes*; es decir, que si uno de ellos se puede resolver en tiempo lineal, entonces los 3 problemas se pueden resolver en tiempo lineal:

Selección: Dado *k* entre 1 y *n*, seleccionar el k-ésimo menor elemento del vector V[1..n].

Mediana: Calcular la mediana del vector V[1..n] donde la mediana es el elemento que ocuparía la posición (*n*+1) div 2 del vector V[1..n] después de ordenarlo.

Descomposición: Descomponer el vector V[1..n] en dos vectores *X* e *Y* de tamaños respectivos (n div 2) y *(n - (n* div 2)), tales que para todo x ∈ *X* e y ∈ *Y* se tiene x≤y.

--------------------------Solución------------------------

Suponiendo que existe un algoritmo lineal para resolver el problema de selección (de hecho, lo hemos visto ya en el Ejercicio I 1.9: selección2. pero aquí nos basta la hipótesis de su existencia), entonces es claro que podemos calcular la mediana en tiempo lineal, ya que se trata de un caso particular del problema de selección: seleccionar el *((n* + 1) div 2)-ésimo elemento.

Disponiendo de un algoritmo lineal mediana para calcular la mediana, podemos resolver el problema tle

selección de forma lineal, siguiendo la misma idea que en el Ejercicio 11.9 para selección!, pero utilizando ahora como pivote para hacer la partición la mediana calculada por mediana:

proc selección3(V[1 ..n] de *elemento*, **e** *c*, *f*, *k*: *nat*, *késimo*: *elemento*)
si *c* = *f* **entonces**
késimo := V[c]
si no
med := mediana(V, c, *f*)
partición(V, c, *f*, *med*, *i*, *j*)
casos
k < *i* → selección3(V, c, *i*-1, *k*, *késimo*)
 i ≤ *k* ∧ *k* ≤ *j* → *késimo* := V[k]
 k > *j* → selección3(V, *j*+1. *f*, *k*, *késimo*)
fcasos
fsi
fproc

Puesto que partición es lineal (véase el Ejercicio 11.8) y, por hipótesis, mediana también lo es. el algoritmo selección3 resulta ser también lineal.

Para resolver el problema de descomposición utilizando mediana (que suponemos lineal), basta calcular la mediana y llamara partición utilizando como pivote dicha mediana. Los elementos en V[1..(n div 2)] serán los elementos que copiemos en *X* y los elementos en V](*n* div 2) + 1 ..n] serán los que copiemos en *Y*.

Por último, veamos cómo calcular la mediana a partir de un algoritmo que realiza la descomposición en tiempo lineal. Si *n* es par, (1 + *n*) div 2 = *n* div

2. por lo que la mediana será el máximo elemento de *X;* y si *n* es impar, (1+*n)* div 2 = (n div 2) + 1 por lo que la mediana será el mínimo elemento de y. Como calcular el máximo y el mínimo también tiene coste lineal, tenemos un método para calcular la mediana en tiempo lineal.

11.11. Elementos pequeños

Dado un vector V[1..n] de elementos que se pueden ordenar, se desea hallar los *m elementos más pequeños*, donde *m* es mucho más pequeño que *n.* ¿Qué es mejor.

a) ordenar V[1..n] y después coger los *m* primeros elementos V[1..m],
b) ir seleccionando el primer elemento, después el segundo, y así hasta el elemento m-ésimo, o
c) utilizar algún otro método?

Generalizar el mejor método para, fijada una posición *p* del vector, hallar los *m* elementos que ocuparían en el vector ordenado las posiciones *p, p*+1...., *p*+*m*-1.

--------------------------Solución--------------------------

El primer método, ordenar V[1..n] y después coger los *m* primeros elementos, tiene la complejidad de ordenar. Si. por ejemplo, utilizamos *ordenación por mezclas,* sabemos que su coste en términos del número de comparaciones está en Θ(n logn).

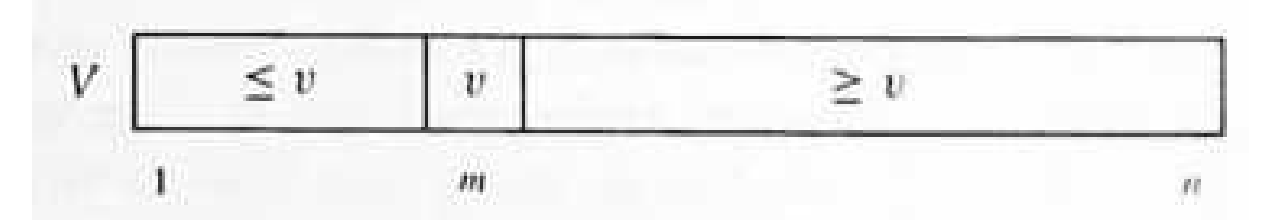
Incluso utilizando un algoritmo de coste lineal para la selección (véase selección2 en la solución del Ejercicio 11.9). el coste del segundo método está en Θ*(mn).*

Un tercer método, más eficiente que los anteriores, sería:

1) llamar a selección2(V,*1,n,m,v).*
2) llamar después a partición(V,1,*n,v,i,j), y*

3)devolver V[1..m].

La primera llamada devuelve en v el m-ésimo menor elemento de *V.* Tras la llamada a partición (véase el Ejercicio 11.8) resulta el vector



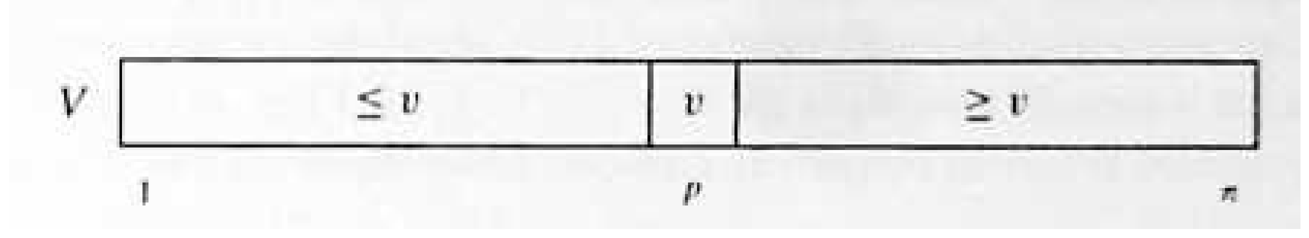
Por tanto, en V[1..m] se encuentran los *m* elementos más pequeños de V, aunque no necesariamente ordenados. El coste de este tercer método está en Θ(n), ya que este es el coste tanto de selección2 como de partición.

En realidad, en la implementación de selección2 en el Ejercicio 11.9 siempre se realiza una llamada a partición antes de devolver el resultado, por lo que el segundo paso sería innecesario.

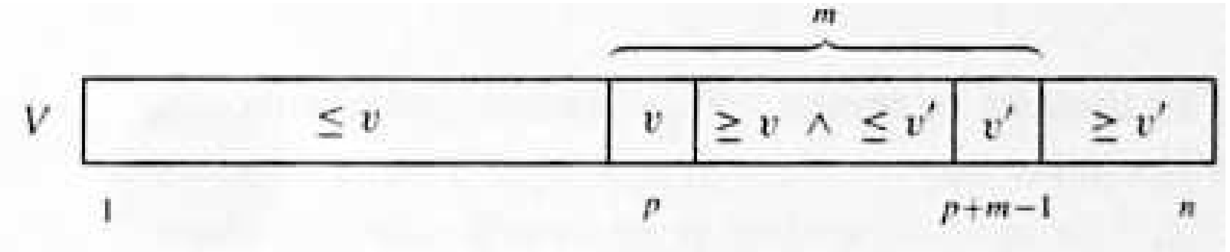
La generalización de este último método se consigue con la siguiente secuencia de acciones:

selección2(V, 1, *n, p,* u)
partición(V, 1, *n, v, i, j)*
selección2(V, *p, n, p*+*m*-1, v')
partición(V,*p*,*n*,*v*',i,j)
devolver *V[p..p*+*m*-1]

La primera llamada a selección2 devuelve en *v* el p-ésimo menor elemento de V, y con la primera llamada a partición el vector resultante es:



La segunda llamada a selección2 devuelve en v' el $(p+m-1)$-ésimo menor elemento de V, entendiendo $(p+m-1)$ como un índice absoluto (entre 1 y n), y con la llamada a partición el vector resultante es ahora:



Con lo que en $V[p..p+m-1]$ se encuentran los m elementos solicitados. Claramente, el coste de este método sigue estando en $\Theta(n)$.

11.12. Mediana

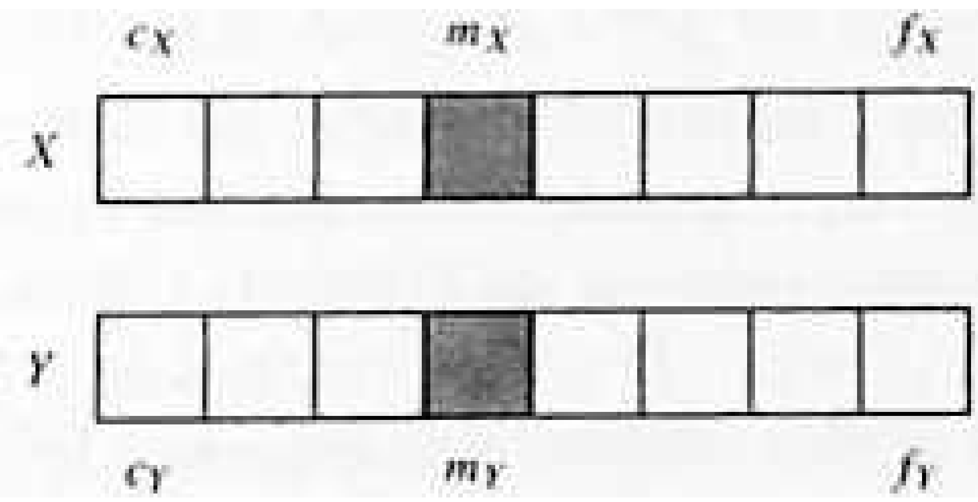
Dados 2 vectores de enteros X[1..n] e Y[1..n] con n≥1, ordenados de forma creciente, escribir un algoritmo para hallar la **mediana** del vector formado por el total de los 2n elementos.

-------------------------Solución-------------------------

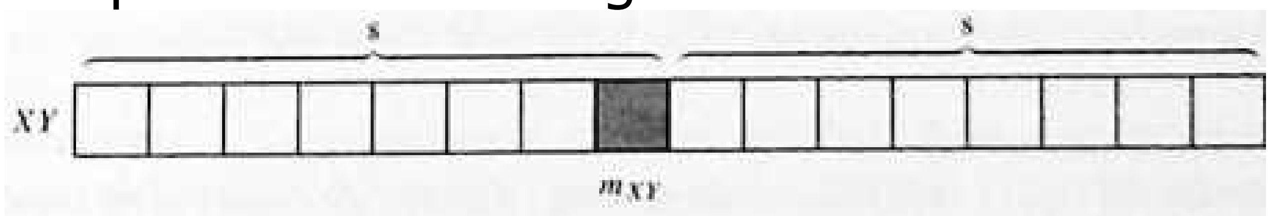
Distinguimos casos según que el tamaño del vector *X* (y el de Y, ya que los dos vectores tienen el mismo tamaño) sea par o impar. Ilustraremos nuestras ideas con un ejemplo.

Caso 1: El tamaño es par.

Supongamos que partimos de los vectores $X[c_X..f_X]$ e $Y[c_Y..f_Y]$ con 8 elementos cada uno. y sean m_X y m_Y, respectivamente, las posiciones de las medianas de *X* e Y.



Al juntarlos obtendremos un vector con 16 elementos, XY, donde la mediana ocupará el octavo lugar.



Si comparamos $X[m_X]$, la mediana de X, con $Y[m_Y]$, la mediana de Y, se tienen los siguientes casos:

(a) Si $\mathbf{X[m_X] = Y[m_Y]}$, hay 6 elementos

menores o iguales que ellos, así que estos elementos ocuparán las posiciones séptima y octava en el vector XY ordenado y, por tanto, la mediana del vector XY es $X[m_X]$ (o $Y[m_Y]$).

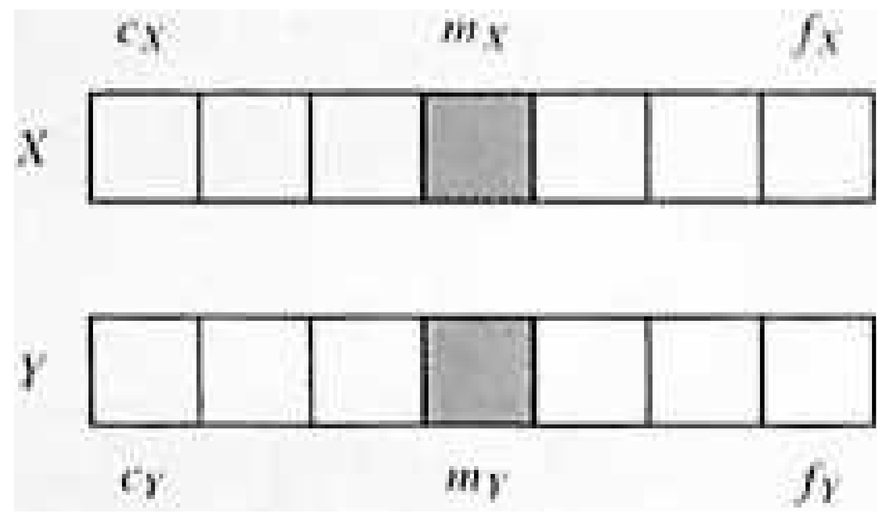
(b) Si **$X[m_X] < Y[m_Y]$,** los elementos en $X[c_x..m_x]$ pueden descartarse, pues todos ellos tienen al menos 9 elementos mayores o iguales, los elementos en $X[m_x+1..f_x]$ y los elementos en $Y'[m_Y..f_Y]$. Lo mismo ocurre con los elementos en $Y[m_Y + 1..f_Y]$. pues estos tienen 8 elementos menores o iguales, los elementos en $y[c_Y..m_Y]$ y los elementos en $X[c_x..m_x]$. En tal caso, podemos pasar a buscar la mediana en $X[m_x+1..f_x]$ e $Y[c_Y..m_Y]$ pues la mediana de *XY* no cambia si quitamos el mismo número de elementos a su izquierda que a su derecha.

(c) Con un razonamiento similar, si **$X[M_X] > Y[m_Y]$** podemos pasar a buscar la mediana en $X[C_X..m_X]$ e $Y[m_Y+1..f_Y]$.

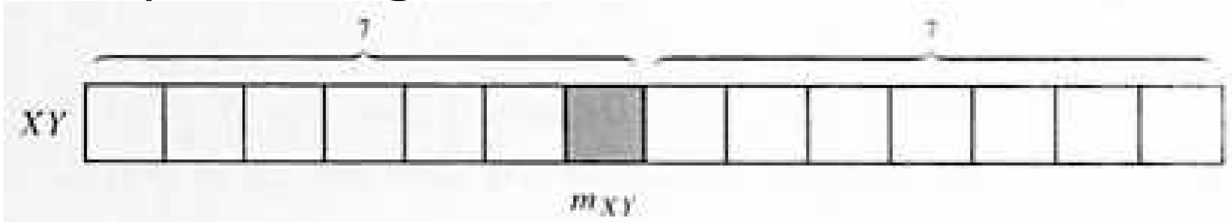
Nótese que en ambos casos, los dos vectores con los que hacemos las llamadas recursivas son del mismo tamaño.

Caso 2: El tamaño es impar.

Supongamos que partimos de los vectores $X[c_X..f_X]$ e $Y[c_Y..f_Y]$ con 7 elementos cada uno.



Al juntarlos obtendremos un vector con 14 elementos, *XY,* ocupando la mediana el séptimo lugar.



Comparamos $X[m_X]$, la mediana de X, con $Y[m_Y]$, la mediana de Y:

(a) Si $X[m_X] = Y[m_Y]$, estos elementos ocuparán las posiciones séptima y octava en el vector *XY* y, por tanto, la mediana del vector *XY es* $X[m_X]$, igual que en el caso par.

(b) Si $X[m_X] < Y[m_Y]$, se pueden descartar los elementos en X[cx..mx – 1] porque tienen al menos 8 elementos mayores o iguales, los elementos en X[mx..fx] y los elementos en y[my..fy]. Otro tanto ocurre con los elementos en Y[my..fy], pues estos tienen 7 elementos menores o iguales, los elementos en $Y[c_Y..m_Y -1]$ y los elementos en $X[cx..m_X]$. Para que la llamada recursiva se haga con vectores del mismo tamaño, descartaremos los elementos en $Y[my+1..f_Y]$, pasando a buscar la mediana en $X[mx..f_X]$ y $Y[c_Y..m_Y]$.

(c) De forma análoga, si X[mX] > Y[mY], podemos pasar a buscar la mediana en X[cX..mX] e Y[mY..fY].

El algoritmo que implementa estas ideas es el siguiente, donde el caso básico es inmediato.

$\{fx - c_X = f_Y - c_Y\}$

```
fun medianaXY(X[1..n]. y[1..n] de ent,
cx,fx,cy,fy: nat) dev mediana: ent
 t := fx - cx + 1      {tamaño de los subvectores}
  si t=1 entonces    {subvectores unitarios}
    mediana := mín(X[cx], Y[cy])  {la mediana es el
    mínimo}
  si no
    mx := (cx + fX) div 2
    my := (cY + fY) div 2
    casos
      X[mX] < y[mY] →
        si par(t) entonces
          mediana := medianaXY(X, Y,mx+1, fX, cY,
          mY)
        sino
          mediana := medianaXY(X, Y, mx, fx, cY,
          mY)
        fsi
      [] X[mX] =Y[mY]→ mediana := X[mX]
      [] X|mX] > Y[mY]→
        si par(t) entonces
          mediana := medianaXY(X, Y, cX, mX, mY+1,
          fY)
        sino
          mediana := medianaXY(X, Y, cX, mX, mY,fY)
        fsi
    fcasos
  fsi
ffun
```

Como se ve, con una única comparación logramos dividir el problema por la mitad, y en todos los casos se realiza una única llamada recursiva. Por tanto, se obtiene un coste en $\Theta(\log n)$.

11.13. Mayoritario

Dado un vector V[1..n] de *n* elementos (no necesariamente ordenables). se dice que un elemento x es ***mayoritario*** en V cuando el número de veces que x aparece en V es estrictamente mayor que *n*/2.

a) Escribir un algoritmo que decida en tiempo *O(n* logn) si un vector V[1..n] contiene un elemento mayoritario y devuelva tal elemento cuando exista.

b) Suponiendo ahora que los elementos del vector V[1..n] se puedan **ordenar**, escribir un algoritmo que decida en tiempo lineal si V[1..n] contiene un elemento mayoritario y devuelva tal elemento cuando exista.

c) Hacer lo mismo que en el apartado anterior, pero **sin** suponer que los elementos se pueden **ordenar**.

----------------------------Solución--------------------------

Apartado (a)

Si el vector V tiene un elemento mayoritario. este tiene que ser mayoritario en al menos una de las mitades del vector. Utilizando esta idea, podemos mirar, de forma recursiva, si existe un elemento mayoritario en cada una de las dos mitades del vector. Si alguna de las mitades contiene mayoritario, contamos cuántas veces aparece este en todo el vector, y si aparece en más de la mitad de las posiciones, será el elemento mayoritario buscado. El algoritmo que busca el

mayoritario de V[c..f] es el siguiente:

```
fun mayoritario1 (V[1..n] de elemento,
c,f:nat) dev (existe:bool, mayor:elemento⟩
  si c=f entonces
    ⟨existe, mayor⟩ := ⟨cierto, V[c]⟩
  si no
    m := (c+f) div 2
    ⟨existe₁, mayor₁⟩ := mayoritario1 (V, c, m)
    ⟨existe₂, mayor₂⟩ := mayoritario1 (V, m+1,f)
    existe := falso
    si existe₁ entonces   {comprobamos el primer
    candidato}
      ⟨existe, mayor⟩ := (comprobar(V, mayor₁,
      c,f), mayor₁⟩
    fsi
    si ¬ existe ∧ existe₂ entonces   {comprobamos
    el segundo candidato}
      ⟨existe, mayor⟩ := ⟨comprobar(V, mayor₂,
      c, f), mayor₂ ⟩
    fsi
  fsi
ffun
```

El predicado comprobar verifica si el candidato a mayoritario efectivamente lo es. Para ello, primero cuenta el número de apariciones del valor candidato en el vector, entre dos posiciones dadas, y después comprueba si el número obtenido es estrictamente mayor que la mitad del número de elementos en el segmento considerado del vector:

```
fun comprobar(V[1..n] de elemento, x:
elemento, c,f:nat) dev válido:bool

 veces := 0
 para i = c hasta f hacer
   si V[i]=x entonces
```

```
    veces := veces + 1
  fsi
 fpara
 válido := veces > (f-c+1) div 2
ffun
```

El tiempo de ejecución *T(n)* de mayoritario1 se describe mediante la recurrencia

$$T(n) = \begin{cases} c_0 & n = 1 \\ 2T(n/2) + c_1 n & n > 1 \end{cases}$$

de donde se deduce que *T(n)* ∈ *Θ*(*n* log n).

CHATGPT

El pseudocódigo proporcionado describe un algoritmo para determinar si existe el elemento **MAYORITARIO** en un vector V[1..n], donde el elemento mayoritario se define como el (único) que aparece más de n/2 veces en el vector. El algoritmo sigue un enfoque de divide y vencerás y tiene una complejidad temporal de O(n log n).

Función `mayoritario1(V[1..n] de elemento, c, f: nat) dev (*existe*: bool, *mayor*: elemento)`

Esta función busca recursivamente un elemento mayoritario en el subvector V[c..f].

1. **Caso Base**: Si el subvector consta de un solo elemento (c = f), este elemento es trivialmente el mayoritario en ese subvector, por lo que la función devuelve `cierto` y el elemento en sí.
2. **División**: Si el subvector tiene más de un elemento, se divide en dos mitades: V[c..m] y V[m+1..f], donde m = (c + f)/2. Luego, la función se llama a sí misma recursivamente para cada mitad.
3. **Combinación**:
   - Si se encuentra un elemento mayoritario `mayor1` en la primera mitad (V[c..m]), se verifica si `mayor1` es mayoritario en el subvector completo V[c..f] utilizando la función `comprobar`.
   - Si no se encuentra un mayoritario en la primera mitad o si `mayor1` no es mayoritario en el subvector completo, se procede a verificar el segundo candidato

`mayor2` (si existe) de la segunda mitad (V[m+1..f]) de manera similar.

Si alguno de los candidatos resulta ser mayoritario en el subvector completo, la función devuelve `cierto` y el elemento mayoritario. En caso contrario, devuelve `falso`, indicando que no hay elemento mayoritario.

Función comprobar(V[1..n] de elemento, x: elemento, c, f: nat) **dev** *válido*: bool

Esta función verifica si un elemento candidato `x` es mayoritario en el subvector V[c..f] contando cuántas veces aparece `x` entre las posiciones `c` y `f`:

1. Inicializa un contador `veces` en 0.
2. Recorre cada elemento del subvector V[c..f]. Si un elemento V[i] es igual a `x`, incrementa `veces`.
3. Después de recorrer el subvector, comprueba si `veces` es mayor que la mitad de la longitud del subvector (f-c+1)/2). Si es así, `x` es un elemento mayoritario en el subvector, y la función devuelve `cierto`. En caso contrario, devuelve `falso`.

El algoritmo explota el hecho de que si un elemento es mayoritario en el vector completo, debe ser mayoritario en al menos una de las mitades del vector. Esta propiedad permite reducir el problema a subproblemas más pequeños de forma recursiva. La combinación de los resultados de las mitades y la verificación final con la función `comprobar` aseguran que, si existe un elemento mayoritario en el vector completo, será identificado por el algoritmo. La complejidad O(n log n) proviene de la naturaleza recursiva del algoritmo, que divide el problema a la mitad en cada nivel de recursión y realiza un recorrido lineal en cada llamada a `comprobar`.

Apartado (b)-----------------------

Si los elementos del vector se pueden ordenar, entonces el elemento mayoritario, si existe, se colocará en la posición central del vector cuando este se ordene. Es decir, si un vector ordenable tiene elemento mayoritario, este coincide con su mediana. Por tanto, podemos utilizar el algoritmo selección2 (véase el Ejercicio 11.9) para buscar la mediana, y después comprobar si es mayoritario.

Como selección2 es un procedimiento que modifica el vector de entrada, el siguiente algoritmo usa un vector auxiliar W en el que se copia V.

```
fun mayoritario2(V[1..n] de elemento)
dev (existe:bool, mayor:elemento)
var W[1..n] de elemento

  W := V
  pm := (n+1) div 2 {posición de la mediana }
  selección2(W, 1, n, pm, mediana)
  ⟨existe, mayor⟩ := ⟨comprobar(W, mediana,
  1, n), mediana⟩
ffun
```

Los costes en tiempo y en espacio de mayoritario2 están en $\Theta(n)$.

Apartado (c)---------------------------

La idea, tomada de [Man89, Sección 6.10], va a ser considerar un *candidato* a mayoritario y recorrer el vector contando cuántas veces aparece dicho candidato más que cualquier otro elemento. Cada vez que el contador llegue a valer 0, cambiaremos de candidato, continuando la cuenta desde la posición en la que nos quedamos. En realidad, lo que se hace es anular entre sí elementos diferentes. Si el vector tiene elemento mayoritario. terminaremos con él como candidato, porque cada vez que reiniciamos el proceso, el elemento en cuestión seguirá siendo mayoritario en la parte restante del vector. Al terminar de recorrer el vector, tendremos que recorrerlo una vez más para comprobar si el último candidato a mayoritario efectivamente lo es.

```
fun mayoritario3(V[1 ..n] de elemento) dev (existe:bool, mayor: elemento)
 candidato := V[1]
 contar := 1      {cuántas veces más ha aparecido el candidato}
  para i=2 hasta n hacer
   si contar=0 entonces   {elegimos nuevo candidato}
     candidato := V[i];
     contar := 1
   si no          {contar > 0}
     si V[i] = candidato entonces
       contar := contar + 1
     si no
       contar := contar -1
     fsi
   fsi
```

fpara
⟨*existe*, *mayor*⟩ := ⟨comprobar(V, *candidato*, 1, *n*), *candidato*⟩
ffun

El coste de mayoritario3 está en $\Theta(n)$ pues simplemente recorremos el vector 2 veces.

11.14. Maximo y mínimo de vector

Sea *V* (1. .*n*] un vector cuyos elementos se pueden comparar entre sí. Suponiendo que *n* es una potencia de 2 mayor que 2, escribir un algoritmo que encuentre el máximo y el mínimo de V[1 ..n] realizando menos de *2n*-3 comparaciones entre elementos.

---------------------------Solución----------------------------

Dado un vector de *n* elementos, se puede encontrar el máximo realizando *n*-1 comparaciones entre elementos, tras lo cual, se puede determinar el mínimo realizando *n*-2 comparaciones adicionales; con lo que el problema se resolvería con un total de *2n*-3 comparaciones entre elementos.

Sin embargo, podemos mejorar el coste si dividimos el vector por la mitad, y buscamos el máximo y el mínimo de cada parte, para finalmente quedarnos con el mayor de los máximos y el menor de los mínimos. Es decir, realizamos la siguiente secuencia de acciones:

⟨*máx1, mín1*⟩ := máx-mín(V[1..n/2])
⟨*máx2*, *mín2*⟩ := máx-mín(V[n/2 + 1..n])
máx := máx(*máx1*, *máx2)*
mín := mín(*mín1*, *mín2*)

Suponiendo que *n* sea par, el número total de comparaciones entre elementos sería: *2(n/2)* - 3 + *2(n/2)* - 3+1 + 1 =2n-4. Así pues, mediante la división del problema en 2 subproblemas de tamaño

la mitad, hemos logrado ahorrar una comparación entre elementos. ¿Hasta cuánto se puede lograr reducir el número de comparaciones a realizar si seguimos con el proceso de división?

El siguiente algoritmo realiza las divisiones de forma recursiva hasta alcanzar el caso básico del subvector con un único elemento, el cual será simultáneamente el elemento máximo y mínimo:

```
fun máx-mín1 (V[1..n] de elemento, c, f: nat) dev ⟨máx. mín : elemento⟩
  si c = f entonces
    ⟨máx, mín⟩ := ⟨V[c], V[c]⟩
  si no
    m := (c + f) div 2
    ⟨máx1, mín1⟩ := máx-mín1 (V, c, m)
    ⟨máx2, mín2 ⟩ := máx-mín1(V, m+1, f)
    máx := máx(máx1, máx2)
    mín := mín(mín1, mín2)
  fsi
ffun
```

Si *T(n)* es el número de comparaciones entre elementos realizadas por el algoritmo máx-mín1 cuando el número de elementos del vector entre *c* y *f* es *n*, *T*(n) se describe con la siguiente recurrencia:

$$T(n) = \begin{cases} 0 & n = 1 \\ 2T(n/2) + 2 & n > 1 \end{cases}$$

Suponiendo que *n* es una potencia de 2, se tiene que

$$T(n) = 2^i T\left(n/2^i\right) + \sum_{j=1}^{i} 2^j.$$

El caso básico se alcanza cuando $n/2^i$ = 1 o, equivalentemente, cuando *i* = log n. En ese caso, tenemos

$$T(n) = 2^{\log n} T(1) + \sum_{j=1}^{\log n} 2^j = 2n - 2$$

Paradójicamente, al iterar el proceso de división, en lugar de reducir más el número de comparaciones lo hemos incrementado. El problema es que hemos dividido “demasiado": si en lugar de dividir hasta alcanzar subvectores unitarios, detenemos el proceso cuando el tamaño es de 2 elementos, se obtiene el siguiente algoritmo:

```
fun máx-mín2(V[1..n] de elemento, c, f : nat) dev ⟨máx,mín : elemento⟩
  casos
     c = f → {máx. mín) := ⟨V[c], V[c]⟩
  [] c+1=f → {hay 2 elementos}
     si V[c] < V[f] entonces
        ⟨máx, mín⟩ := ⟨V[f], V[c]⟩
     sino
        ⟨máx,mín⟩ := ⟨V[c], V[f]⟩
     fsi
  [] c+1 < f →
     m := (c + f) div 2
     {máx1, mín1⟩ := máx-mín2( V, c, m)
     {máx2, mín2⟩ := máx-min2(V, m+1, f)
     máx := máx(máx1, máx2)
     mín := min(mín1, mín2)
  fcasos
ffun
```

La recurrencia que describe el número de comparaciones entre elementos realizadas por máx-min2 es:

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2T(n/2) + 2 & n > 2 \end{cases}$$

De nuevo, suponiendo que *n* es una potencia de 2, se tiene que

$$T(n) = 2^i T\left(n/2^i\right) + \sum_{j=1}^{i} 2^j.$$

pero ahora, para *n* > 2, el caso básico al que se llega es 2 (en vez de 1), que se tiene cuando $n/2^i = 2$ o, equivalentemente, cuando *i* = log n - 1. De esta forma, tenemos para *n > 2*

$$T(n) = 2^{\log n - 1} T(2) + \sum_{j=1}^{\log n - 1} 2^j$$

$$= \frac{n}{2} + 2^{\log n} - 2 = \frac{3}{2}n - 2$$

$$< 2n - 3.$$

11.15. Poker

Juan es aficionado al juego y todas las noches va a jugar una partida de poker. Para controlar sus pérdidas, anota lo que gana o pierde cada noche como un número entero de euros. Por ejemplo, la siguiente tabla muestra los resultados cosechados por Juan en el último mes:

					29	-7
14	21	30	-47	1	7	-39
23	-20	-36	-41	27	-34	7
48	35	-46	-16	32	18	5
-33	27	28	—22	1	-20	-42

Se observa que empezó bien el mes con una ganancia de 29 euros, pero terminó con una pérdida de 42 euros. El beneficio total obtenido a lo largo de todo el mes es -50 euros. Analizando esta información en retrospectiva, Juan se da cuenta de que si hubiera empezado a jugar el día 16 y terminado el día 26, habría maximizado sus ganancias, obteniendo 105 euros, una diferencia de 155 euros con respecto a su situación actual.

Dado un vector de ganancias/pérdidas de longitud n, se desea encontrar el subvector sobre el cual se consigue el beneficio total máximo.

(a) Desarrollar un algoritmo iterativo de coste $\Theta(n^2)$.

(b) Desarrollar un algoritmo de tipo divide y vencerás de coste $\Theta(n \log n)$.

(c) Desarrollar un algoritmo de tipo divide y vencerás de coste $\Theta(n)$.

(d) Desarrollar un algoritmo iterativo de coste Θ(n).

------------------------Solución------------------------

Apartado (a)-------------------------

Si tenemos que buscar en un vector de enteros V[1..n] el subvector (formado por elementos consecutivos) cuya suma sea máxima, la idea obvia (aplicando la fuerza bruta) consiste en probar todas la posibilidades, es decir, considerar la suma de cada subvector no vacío V[i..j] para $1 \leq i \leq j \leq n$. más el subvector vacío (de suma 0) y quedarse con el mejor. Si la suma de cada subvector se calcula con un tercer bucle, el coste total en tiempo estará en $\Theta(n^3)$; pero si la suma se calcula de forma incremental mediante un acumulador bastan 2 bucles, por lo que el coste es el que se solicita.

Representaremos un subvector mediante un registro con 3 campos: los 2 primeros para indicar el comienzo y el final, y el tercero para guardar la suma total de sus elementos.

```
tipos
   subvector = reg
                 ini : nat
                 fin : nat
                 suma : nat
               freg
ftipos
```

El algoritmo de coste cuadrático que calcula el subvector óptimo es el siguiente:

```
fun subvectorOptimo-it1(V[1..n] de ent)
dev óptimo : subvector
óptimo.ini := 1;
óptimo.fin := 0;
óptimo.suma := 0                          {subvector vacío}
para i = 1 hasta n hacer
    suma := 0
    para j = i hasta n hacer
        suma := suma + V[j]
        si suma > óptimo.suma entonces
            óptimo.ini := i ;
            óptimo.fin := j ;
            óptimo.suma := suma
        fsi
    fpara
fpara
ffun
```

Apartado (b)--------------------

Para obtener un algoritmo recursivo más eficiente, vamos a aplicar el método de ***divide y vencerás***, como sugiere el enunciado. La idea consiste en partir el vector por la mitad para buscar el subvector de suma máxima en cada mitad, obteniendo uno en la primera parte, *izq.* y otro en la segunda, *der.* Los subvectores del vector inicial que no se han considerado en las llamadas recursivas son todos aquellos que pasan por el medio. De modo que tenemos que calcular además el subvector de suma máxima que pasa por el centro, *cen.* Basta ahora escoger el mejor entre *izq, der y cen.*

Consideramos como caso básico el vector con un único elemento. Entonces

el subvector de suma máxima estará formado por ese elemento, si es positivo, siendo el subvector vacío en otro caso. La llamada inicial es subvectorOptimo-dv1(V, 1, *n*).

```
fun subvectorOptimo-dv1(V[1..n] de ent, c,f:
nat) dev optimo: subvector
var izq, der, cen: subvector

si c = f entonces
  si V[c] >0 entonces
    optimo.ini := c ;
    optimo.fin := f ;
    optimo.suma := V[c]
  si no      {devolvemos el subvector vacío}
    optimo.ini := c+1;
    optimo.fin := f ;
    optimo.suma := 0
  fsi
si no
  m := (c+f) div 2
  izq := subvectorOptimo-dv1 (V, c, m)
  der := subvectorOptimo-dv1 (V, m+1, f)
  cen := subvectorCentral(V, c, f)
  optimo := maxSubvector(maxSubvector(izq,
  der), cen)
fsi
ffun
```

Dados dos *subvectores,* la función maxSubvector, que no detallamos, devuelve el subvector con valor máximo en el campo *suma.*

El siguiente algoritmo calcula el subvector de suma máxima que pasa por el centro. Para ello extiende lo máximo posible un segmento desde el centro, primero hacia la izquierda y luego hacia la derecha. Nótese que para que un subvector que pase por el centro tenga

suma máxima también tienen que serlo sus sumas parciales hacia la izquierda y hacia la derecha desde el centro.

```
fun subvector-central(V[1..n] de ent, c,f: nat)
dev optimo : subvector

 m := (c+f) div 2
 {parte izquierda}
 suma := 0; izqMáx := 0; optimo.ini := m+1
  para k=m hasta c paso - 1 hacer
   suma := suma + V[k]
   si suma > izqMax entonces
     izqMax := suma ;
     optimo.ini := k
   fsi
  fpara

 {parte derecha}
 suma := 0; derMax := 0; optimo.fin := m
  para k=m+1 hasta f hacer
   suma := suma +V[k]
   si suma > derMax entonces
     derMax := suma ;
     optimo.fin := k
   fsi
  fpara
 optimo.suma := izqMax + derMax
ffun
```

El coste en tiempo de este último algoritmo es lineal con respecto a la longitud del vector, pues se recorre todo el vector (en dos etapas). Por tanto, en el algoritmo recursivo hacemos 2 llamadas recursivas de tamaño la mitad más un proceso adicional de coste lineal, con lo que su coste en tiempo está en Θ(n logn).

Apartado (c)--------------------

El algoritmo recursivo del apartado anterior no aprovecha toda la potencia del método *divide y vencerás*, puesto que al calcular el subvector óptimo central tiene que volver a sumar los elementos de las partes izquierda y derecha correspondientes, lo que supone un gasto lineal adicional en cada llamada recursiva. Se puede evitar esto si la llamada con el subvector izquierdo nos devuelve además el subvector óptimo que alcance su extremo derecho y, análogamente, la llamada con el subvector derecho nos devuelve también el subvector óptimo que comience justo en su extremo izquierdo. De esta forma, el subvector central óptimo correspondería a la concatenación de estos dos nuevos subvectores. Esto nos obliga a generalizar la función subvector-óptimo-dv1 para que devuelva 3 subvectores óptimos: el solicitado (*ópt*). el que comienza en el extremo izquierdo *(izq).* y el que comienza en el extremo derecho *(der).* Sin embargo, para determinar *izq* no es suficiente con el resultado obtenido para la llamada recursiva con la parte izquierda del vector, puesto que *izq* podría ir más allá de la posición central del vector. En ese caso *izq* sería la concatenación de toda la parte izquierda, con el subvector óptimo izquierdo obtenido para la parte derecha. Análogamente sucede para determinar

der. Es necesario, por tanto, conocer la suma total de los elementos de cada parte del vector, y esto se puede conseguir eficientemente de la forma siguiente:

```
fun subvector-óptimo-dv2(V[1..n] de ent, c, f : nat) dev ⟨ópt, izq, der: subvector, total: nat⟩
var ópt1, izq1, der1, ópt2, izq2, der2, aux, aux' : subvector
si c = f entonces
   si V[c] > 0 entonces
      aux.ini := c ;
      aux.fin := c ;
      aux.suma := V[c];
      ⟨opt, izq, der, total⟩ := ⟨aux, aux. aux, V[c]⟩;
   si no
      aux.ini := c+1;
      aux.fin := c;
      aux.suma := 0;
      aux'.ini := c;
      aux'.fin := c-1;
      aux'.suma :=0;
      ⟨opt, izq, der, total⟩ := ⟨aux aux', aux, V[c]⟩
   fsi
si no
   m := (c + f) div 2;
   ⟨ ópt1, izq1, der1, total1 ⟩ := subvector-óptimo-dv2( V, c, m);
   (ópt2, izq2. der2, total2) := subvector-óptimo-dv2(V, m+1, f);
   aux.ini := der1.ini;
   aux.fin := izq2fin;
   aux.suma := der1.suma + izq2.suma;
   ópt := máx-subvector(máx-subvector(opt1, ópt2), aux);
   aux.ini := c;
   aux.fin := izq2.fin;
   aux.suma := total1 + izq2.suma;
   izq := máx-subvector(izq1, aux);
   aux.ini := der1.ini;
   aux.ini := f;
```

```
        aux.ini := total2 + der1.suma;
        der := máx-subvector(der2, aux);
        total := total1 + total2;
    fsi
ffun
```

Puesto que ahora el coste de dividir en subproblemas y componer las soluciones es constante, el coste total del algoritmo es lineal respecto al número de elementos del vector.

Apartado (d)--------------------

La idea del algoritmo iterativo de coste lineal consiste en hacer un único recorrido del vector de izquierda a derecha manteniendo en cada momento cuál es el subvector de suma máxima de la parte recorrida, *óptimo.* Cuando en el recorrido pasamos al siguiente elemento, los nuevos subvectores que aparecen son todos los que contienen a dicho elemento. De todos estos nuevos subvectores, el que tenga suma máxima puede mejorar el valor de *óptimo.* Por tanto, además del subvector de suma máxima de la parte recorrida necesitamos mantener el *sufijo* (subvector que acaba en el último elemento recorrido) de suma máxima de dicha parte. Nótese que el sufijo de suma máxima puede ser vacío cuando todos los sufijos tienen suma negativa. El algoritmo resultante es el siguiente:

fun subvector-óptimo-it2(V'[l..n] **de** ent) **dev** *óptimo : subvector*
var *sufijo : subvector*
óptimo.ini := 1 ; *óptimo.fin* := 0 ; *óptimo.suma* := 0 (subvector vacío)
sufijo.ini := 1 ; *sufijo.fin* := 0; *sufijo.suma* := 0 (el sufijo máximo es vacío i
para *i* = 1 **hasta** *n* **hacer**
si *sufijo.suma* + V[i] > 0 **entonces** { el sufijo crece)
sufijo.suma := *suma* + V[í] ; *sufijo.fin* := *i*
si no (el sufijo se hace vacío)
sufijo.ini := *i* + 1 ; *sufijo.fin* := *i* : *sufijo.suma* := 0
fsi

si *sufijo.suma* > *óptimo.suma* **entonces** (se mejora *óptimo* }
óptimo := *sufijo*
fsi
fpara
ffun

11.16. **a+b=S** **(Febr2023)**

Dado un vector *C*[1..n] de números enteros distintos, y un número entero **S**:

(a) Diseñar un algoritmo de complejidad Θ(n logn) que determine si existen o no 2 elementos de *C* tales que su **suma** sea exactamente ***S***.

(b) Suponiendo ahora *ordenado* el vector *C*, diseñar un algoritmo que resuelva el mismo problema en tiempo Θ(n).

---------------------Solución-----------------

Apartado (a)----------------

Podemos resolver el problema siguiendo la siguiente idea:

1. ordenar el vector C, y
2. para cada elemento x de *C* buscar, utilizando búsqueda binaria (Ejercicio 11.1), el entero *S - x.* Si alguna de las búsquedas tiene éxito, habremos encontrado 2 elementos de C que suman *S.*

Utilizando, por ejemplo, la *ordenación por mezclas,* el coste en tiempo del primer paso está en Θ(n log n), y el coste del segundo paso está también en Θ(n log n). ya que, en el caso peor, se hacen *n* búsquedas binarias, de coste en Θ(log n) cada una de ellas. Por tanto el coste total del algoritmo está en **Θ(n logn)**. como se solicitaba.

El algoritmo resultante es el siguiente:

```
fun buscar-dos(C[1..n] de ent, S:ent) dev
⟨existe:bool, x1, x2: ent⟩
var V[1..n] de ent

V := C;
ordenar(V)
j := 1;
existe := falso
```

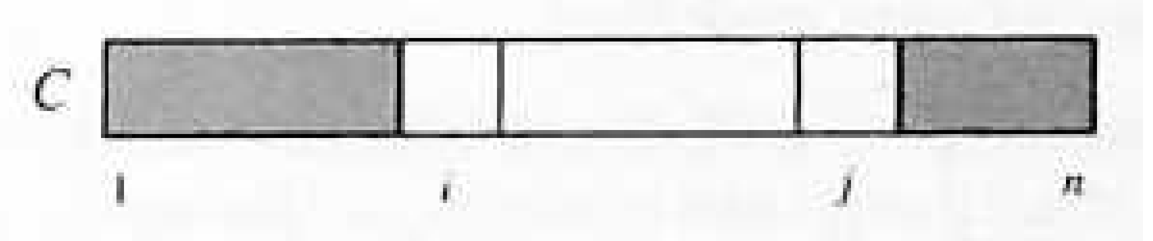
```
mientras j ≤ n ∧ ¬ existe hacer
    x1 := V[j];
    x2 := S-x1
    ⟨existe, p⟩ := búsqueda-binaria(V, x2, 1, n)
    existe := existe ∧ p ≠ j {no puede usarse el mismo
    número 2 veces }
    j := j+1
fmientras
ffun
```

Para no modificar los datos de entrada al ordenar los números, los hemos copiado en un vector auxiliar *V* que es el que se ordena y sobre el que se realizan las búsquedas binarias. En este caso no podemos utilizar la ordenación de índices dada en el Ejercicio 11.6 porque entonces necesitaríamos modificar también la búsqueda binaria para que accediera a los valores a través del vector de índices.

Apartado (b)----------------------------

Suponiendo que *C* está ya ***ordenado*** de manera creciente, si siguiésemos con el segundo paso del apartado anterior tendríamos una complejidad Θ(n logn). superior a la que nos piden. Tenemos que usar otra idea.

En este caso recorreremos el vector *C* con dos índices *i* y *j* que apuntarán, respectivamente, al menor y mayor elemento de C "no descartados". Inicialmente tomamos *i*=1 y *j*=n.



En cada caso, compararemos la suma

C[i]+C[j| con *S.*

- Si C[i] + C[*j*] = S, habremos encontrado el par de números buscado.
- Pero si C[i]+ C[j] < S podemos descartar C[i] porque al sumarlo con cualquier otro elemento no descartado (entre C[i+1] y C[j-1]) seguiremos obteniendo algo menor que S ya que el vector está ordenado.
- Igualmente, si C[i] + C[y] > S podremos descartar C[j] ya que al sumarlo con cualquier otro elemento no descartado (mayor que C[i]) seguiremos obteniendo una suma mayor que S.

El algoritmo que implemento estas ideas, con un coste en tiempo en Θ(n) es el siguiente:

{C[1]≤...≤C[n]}

```
fun buscar-dos-ordenado(C[1..n] de ent, S : ent) dev ⟨existe : bool, x1,x2 : ent⟩
i := 1;
j := n;
existe := falso
  mientras i < j ∧ ¬existe hacer
    casos
      C[i] + C[j] < S → i := i + 1
      C[i] + C[J] = S → (existe,x1,x2) := (cierto, C[í], C[j])
      C[i] + C[j] > S → j:=j-1
    fcasos
  fmientras
ffun
```

11.17. Multiplicar binarios

Desarrollar un algoritmo eficiente para **multiplicar** 2 números naturales **binarios** de *n* dígitos, suponiendo que disponemos de las siguientes operaciones básicas, todas ellas de coste constante:

- producto de bits.
- suma de bits (con acarreo).
- desplazamientos de bits una posición (a la izquierda y a la derecha).

----------------------------Solución----------------------------

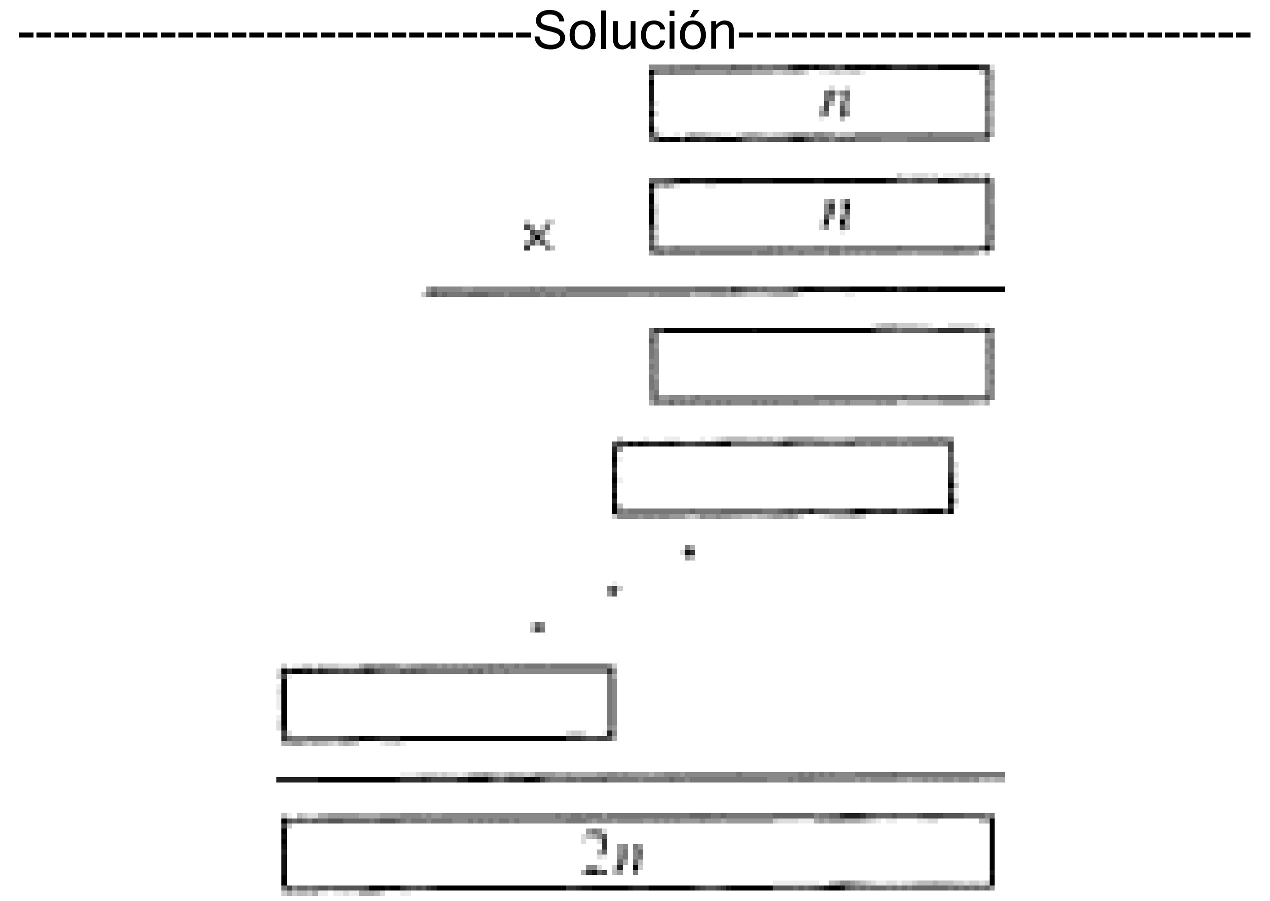


Figura 11.5: Esquema para multiplicar dos naturales.

El algoritmo de multiplicación "clásico" se describe por medio de la Figura 11.5, donde la fila í-ésima (para *i* entre 1 y la longitud del multiplicador) implica el siguiente trabajo:

1. multiplicar el multiplicando por el bit í-ésimo del multiplicador.
2. desplazar la fila *i*- 1 posiciones a la izquierda, y
3. sumar con el resultado del paso anterior (para hacer las sumas incrementalmente).

Cada una de estas acciones tiene un coste

lineal con respecto al número de bits, y como tenemos *n* filas en total, obtenemos un coste cuadrático.

Vamos a intentar mejorar el algoritmo utilizando divide y vencerás. Para ello, dividimos cada operando en dos partes:

$$b_1 = A2^{n/2} + B$$

$$b_2 = C2^{n/2} + D$$

De esta forma

$$b_1 b_2 = AC2^n + (AD + CB)2^{n/2} + BD,$$

con lo que hemos pasado de un problema de tamaño n a 4 problemas de tamaño *n/2.* En el caso básico los números son de un bit. Se tiene entonces la siguiente recurrencia:

$$T(n) = \begin{cases} c_0 & n = 1 \\ 4T(n/2) + c_1 & n > 1 \end{cases}$$

Y, por tanto, $T(n) \in \Theta(n^2)$.

De momento, no hemos mejorado la complejidad teórica. ¿Podemos reducir de alguna manera el número de llamadas recursivas? En realidad, en *AD* + *CB* no nos interesan tanto los productos intermedios como la suma de ambos. ¿Podernos obtener el resultado final sin hacer los dos productos? La respuesta es afirmativa, ya que:

$AD + CB = (A - B)(D - C) + BD + AC,$

que implica realizar un único producto nuevo, ya que los otros dos ya los calculamos **de todas formas** (además ahora hemos de realizar algunas sumas y restas adicionales, de bajo costel. De esta forma, la recurrencia obtenida es

$$T(n) = \begin{cases} c'_0 & n = 1 \\ 3T(n/2) + c'_1 & n > 1 \end{cases}$$

y. por tanto $T(n) \in \Theta\left(n^{\log_2 3}\right) = \Theta(n^{1.59})$.

Hay que arreglar todavía un pequeño detalle, y es que *A-B y D-C* pudieran dar resultados negativos, por lo que habrá que utilizar entonces números binarios con signo. Otra posibilidad sería partir de *(A+B)(C+*D). pero entonces las sumas podrían resultar de tamaño $\frac{n}{2} + 1$.

Para realizar el algoritmo esbozado representamos cada número binario en un vector de *n* posiciones, y el resultado será, en general, de tamaño *2n.* Supondremos, sin pérdida de generalidad, que n es potencia de 2. Aunque el algoritmo de multiplicación es para números positivos, ya se ha comentado que algunos de los valores intermedios pueden ser negativos. Por ello, para los valores intermedios se considerará el vector ampliado a la posición 0 para mantener el bu de signo (0 positivo. 1 negativo). El algoritmo que implemento estas ideas es el siguiente:

```
fun multiplicar-bin(B1[1..n], B2[1..n] de 0..1, c,
f: nat) dev B[1..2n] de 0..1
var X1[0..2n], X2[0..2n], X3[0..2n], X4[0..2n],
Y1[0..n], Y2[0..n] de 0..1
  si c = f entonces
    B[1] := producto-bit(B1[c], B2[c]);
  si no
    m := (c + f) div 2;
    t := m - c + 1;
    X1[1..2t] := multiplicar-bin(B1, B2, C, m);
    X1[0] := 0   {BD}
    X2[1..2t] := multiplicar-bin(B1, B2, m+1, f);
```

```
X2[0] := 0   {AC}
Y1[0..t] := restar-bin(B1[m+1..f], B1[c..m])
{A – B}
Y2[0..t] := restar-bin(B2[c..m], B2[m+1..f])
{D-C}
X3[1..2t] := multiplicar-bin(Y1[1..t], Y2[1..t],
1, t)   {(A-B)(D-C)}
X3[0] := (Y1[0] + Y2[0] mod 2;
X4[1..2t] := sumar-bin(X1, X2, X3);
X4[0] := 0;  {AD + CB}
B[1..4t] := sumar-bin(desplazar(X2, 2t),
desplazar(X4, t), X1);
fsi
ffun
```

Se suponen implementadas las funciones producto-bit (para multiplicar dos bits), resta-bin (para restar dos números binarios sin signo, obteniendo un resultado *con signo).* suma-bin (para sumar varios números binarios con signo, obteniendo un resultado *sin signo)* y desplazar (para desplazar los bits hacia la izquierda, o posiciones más significativas).

11.18. Multiplicar complejos

Desarrollar un algoritmo para multiplicar *n* números **complejos**, realizando solamente 3(n-1) multiplicaciones entre reales.

------------------------Solución--------------------------

Para multiplicar el número complejo *a*+*bi* por c+*di* operamos así:

$$(a+bi)(c+di)= ac + (ad+bc)i - bd.$$

realizando 4 multiplicaciones entre reales. Pero podemos darnos cuenta de que

$$(a-b)(c+d) = ac + ad - bc - bd.$$

de donde

$$ac-bd = (a-b)(c+d) + bc - ad.$$

y, por tanto:

$$(a+bi)(c+di) = \underbrace{(a-b)(c+d)+bc-ad}_{\text{real}} + \underbrace{(ad+bc)}_{\text{imag.}}\,i.$$

En definitiva, para multiplicar 2 números complejos solo es necesario hacer 3 multiplicaciones entre reales:

bc

ad

(a-b)(c+d).

Ahora, para multiplicar n números complejos habrá que realizar *n*-1 multiplicaciones de 2 números complejos, por lo que en total serán 3(n-1) multiplicaciones entre reales.

11.19. x^n

a) Dado un valor numérico x fijo, escribir un algoritmo para calcular x^n con un coste en *O* (log n), en términos del número de multiplicaciones efectuadas.

b) Siendo *F* la matriz $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ consideremos el producto del vector (*i* *j*) por la matriz *F*. ¿Cuál es el resultado cuando *i* y *j* son números consecutivos de la sucesión de Fibonacci?

c) Utilizando las ideas de los apartados anteriores, desarrollar un algoritmo para calcular el n-ésimo número de Fibonacci con un coste en O(log n), en términos del número de operaciones aritméticas efectuadas.

------------------------Solución--------------------------

Apartado (a)

El algoritmo para calcular la potencia x^n en tiempo Θ(log *n*) es bastante conocido;

```
fun potencia (x: número, n: nat) dev y: número
  casos
      n = 0 → y:= 1
    □ n = 1 → y:= x
    □ n > 1 ∧ par(n) → {xⁿ = x^(2(n div 2)) = (x^(ndiv2))²}
        y:= potencia(x, n div 2) ; y:= y * y;
    □ n > 1 ∧ impar(n) → {(xⁿ = x^(2(n div 2)+1) = x(x^(ndiv2))²)}
        y:= potencia(x, n div 2) ; y:= x * (y * y)
  fcasos
ffun
```

Apartado (b)-----------------------------

El producto del vector (*i* *j*) y la

matriz *F* es:

$$\begin{pmatrix} i & j \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} j & i+j \end{pmatrix}$$

Si *i y j* son números consecutivos de la sucesión de Fibonacci, por ejemplo f_{n-1} y f_n, tenemos

$$\begin{pmatrix} f_{n-1} & f_n \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} f_n & f_{n-1}+f_n \end{pmatrix}$$
$$= \begin{pmatrix} f_n & f_{n+1} \end{pmatrix},$$

es decir, que multiplicando 2 números consecutivos de la serie de Fibonacci por la matriz *F* conseguimos el siguiente número de la serie.

Apartado (c)-------------------------------

Veamos cómo podemos generar los números de Fibonacci con ayuda de esta matriz. Tenemos, como casos básicos,

$f_0 = 0$

$f_1 = 1$

Para conseguir f_2 tenemos que multiplicar el vector formado por f_0 y f_1 por la matriz *F y* quedamos con la segunda componente del vector resultado:

$$\begin{pmatrix} 0 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \underbrace{1}_{f_2} \end{pmatrix}$$

Y podemos obtener f3 en la forma

$$\begin{pmatrix} 1 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & \underbrace{2}_{f_3} \end{pmatrix}$$

En general, para obtener f_n podemos hacer

$$\begin{pmatrix} 0 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-1} = \begin{pmatrix} f_{n-1} & f_n \end{pmatrix}$$

De hecho, no es necesario multiplicar por el vector (0 1), pues para todo $n \geq 2$,

$$F^{n-1} = \begin{pmatrix} f_{n-2} & f_{n-1} \\ f_{n-1} & f_n \end{pmatrix}$$

como se puede comprobar fácilmente por inducción sobre *n.* Por tanto, para calcular el n-ésimo número de Fibonacci, basta calcular la potencia (n - 1)-ésima de *F y* coger la componente apropiada.

Una posibilidad es representar las matrices mediante 4 variables y realizar los cálculos apropiados; pero como todas las potencias de *F* son matrices simétricas de la forma $\begin{pmatrix} a & b \\ b & c \end{pmatrix}$ con c = *a* + *b.* basta representar las matrices con solo dos variables de tipo *nat.*

El algoritmo resultante para calcular el n-ésimo número de Fibonacci es el siguiente:

```
fun fibo(n : nat) dev f : nat
  si n < 1 entonces f := n
  si no
     ⟨a, b⟩ := potF(n - 1)
     f := a + b
fsi
ffun
```

Para el algoritmo potF utilizamos la idea del Apartado (a) con la representación mencionada de las matrices. Tenemos que elevar la matriz *F* a un exponente. Veamos cómo multiplicamos una matriz de la forma apropiada por sí misma, y cómo multiplicamos este resultado por la matriz F.

$$\begin{pmatrix} a & b \\ b & a+b \end{pmatrix}\begin{pmatrix} a & b \\ b & a+b \end{pmatrix} = \begin{pmatrix} a^2+b^2 & ab+b(a+b) \\ ba+(a+b)b & b^2+(a+b)^2 \end{pmatrix}$$

donde, teniendo en cuenta solo las operaciones de la primera fila, son necesarias 3 sumas y 4 productos.

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} a & b \\ b & a+b \end{pmatrix}^2 = \begin{pmatrix} ba+(a+b)b & b^2+(a+b)^2 \\ a^2+b^2+ba+(a+b)b & ab+b(a+b)+b^2+(a+b)^2 \end{pmatrix}$$

donde (en la primera fila) son necesarias 4 sumas y 4 productos.

El algoritmo potF es el siguiente:

```
fun potF(n: nat) dev ⟨a,b : nat⟩
 casos
    n = 0 → ⟨a,b⟩ := ⟨1,0⟩
  [] n = 1 → ⟨a,b⟩ := ⟨0,1⟩
  [] n > 1 ∧ par(n) →
    ⟨a,b⟩ := potF(n div 2)
    ⟨a,b⟩ := ⟨a*a + b*b, a*b + b*(a + b)⟩
  [] n > 1 ∧ impar(n) →
    ⟨a,b⟩ := potF(n div 2)
    ⟨a,b⟩ := ⟨b*a + (a+b)*b, b*b + (a+b)*(a+b)⟩
 fcasos
ffun
```

Ya que, en uno y otro caso, el número de operaciones aritméticas para hacer la combinación es constante. pongamos como mucho c_1, la recurrencia que expresa el coste de potF es:

$$T(n) = \begin{cases} c_0 & n \leq 1 \\ T(n/2) + c_1 & n > 1 \end{cases}$$

De donde $T(n) \in \Theta(\log n)$. De aquí se deduce que el coste de fibo(n), en términos del número de operaciones aritméticas elementales, está en $\Theta(\log n)$.

11.20. Moneda falsa

Mr. Scrooge ha cobrado una antigua deuda, recibiendo una bolsa con n monedas de oro. Su olfato de usurero le asegura que una de ellas es falsa, pero lo único que la distingue de las demás es su peso, aunque no sabe si este es mayor o menor que el de las otras. Para descubrir cuál es la falsa. Mr. Scrooge solo dispone de una balanza con dos platillos para comparar el peso de dos conjuntos de monedas. En cada pesada lo único que puede observar es si la balanza queda *equilibrada,* si pesan más los objetos del platillo de la *derecha* o si pesan más los de la *izquierda.*

(a) Demostrar que, en el caso peor, y para $n \geq 3$. hacen falta un mínimo de $\log_3(2n)$ pesadas para determinar cuál es la moneda falsa y si pesa más o menos que las auténticas.

(b) Diseñar un algoritmo eficiente para encontrar la moneda falsa y decidir si pesa más o menos que las auténticas.

--------------------------Solución-----------------------

Apartado (a)

Cada pesada produce uno de 3 posibles resultados (izquierda, equilibrado, derecha), por lo que el *árbol de decisión* correspondiente es un árbol ternario, con un máximo de 3^p hojas para una profundidad *p.* Ahora bien, cada una de las *n* monedas puede ser la falsa, y además puede pesar más o menos que las auténticas. Eso quiere

decir que hay *2n* posibles soluciones y, por tanto, necesitamos al menos *2n* hojas en el árbol de decisión. Consiguientemente, la profundidad mínima de dicho árbol ha de ser $\log_3(2n)$.

Apartado (b)---------------------------------

Para encontrar la moneda falsa haremos lo siguiente: primero dividimos el montón de monedas en 2 montones, *X* e *Y*, de aproximadamente el mismo tamaño, asegurándonos de que el número de monedas en *X* sea par. porque distribuiremos la mitad de las monedas de *X* en un platillo de la balanza, y la otra mitad en el otro platillo. Si la balanza se desequilibra, significa que la moneda falsa está en el montón *X,* pero si está equilibrada, habrá que buscarla en el montón f. Además del caso básico de 3 monedas (donde necesitaremos 2 pesadas para resolverlo), habrá que resolver de forma particular los casos en que *X* o *Y* tengan menos de 3 monedas y haya que buscar en ellos la moneda falsa.

Supondremos las *n* monedas dadas en un vector *M[* l..n], y para representar las pesadas en la balanza utilizaremos una función pesarte;./, *cd.fd).* donde *ci* y fi corresponden a las posiciones de comienzo y final en el vector de monedas del montón para el platillo izquierdo *(cd y fd.* respectivamente, a

las del platillo derecho). Esta función devuelve uno de los tres valores posibles: izq. si se inclina hacia el platillo izquierdo, der. hacia el derecho, y eq si ambos platillos tienen el mismo peso. El algoritmo es el siguiente:

tipos
peso = {ligera, pesada}
balanza = { izq, eq, der}
ftipos

{1≤c≤f≤n ∧ f-c≥2}

fun moneda-falsa(M[1..n] **de** *moneda*, c, *f: nat)* **dev** ⟨**i:** *1..n, p : peso*⟩

m := f - c + 1 (cantidad de monedas)
casos
ni = 3 —
P := pesar(c, *c. c* + 1. c + I) { pesar primera y segunda)
P2 := pesar(c, *c, c* + 2. c + 2) (pesar primera y tercera |
casos
p = izq A *P2* = izq —> *(i. p}* (c. pesada)
üp\=izq A *pi* = eq—» *(i. p)* := (c + I. ligera)
▫ *p*\=eq *A pi* = izq -» *{i. p)* '■=*(c* + 2, ligera)
ü *pt*-eqA*p,*= der —> (i. p):=(<? + 2, pesada)
▫ *p\=derA pi* = eq -> *(i. p)* := (c + I. pesada)
ü *p* = der A *p2* = der -> (i, *p)* := (c. ligera)
fcasos
 ni = 4 —>
pi := pesarte, c, c + I. c + 1) (pesar primera y segunda)
si *p*\ = eq **entonces** { primera y segunda

son auténticas }
(;. p) := moneda-falsa(A/[l..n], *c* + 1. *f)*
si no | tercera y cuarta son auténticas)
ti. p) := moneda-falsa(A/[I..n], e. *f* - 1)
fsi (en cada caso, quitamos una moneda auténtica)
0 *ni* = 5 —»
p := pesarte, *c* + 1. *e* + 2. *c* + 3) { pesar primera y segunda con tercera y cuarta I
si *p* = eq entonces (la falsa es la quinta)
P2 := pesarte, c. *f. f)* (comparar quinta con primera }
si *P2* = izq entonces *{i. p)* := (/.ligera) sino (i. *p*) := (/. pesada) fsi
si no (buscar entre las 4 primeras)
(í.p) := moneda-falsa(A/[l..n|. e. / — 1) **fsi**
0 ;;; = 6 —>
Pi := pesarte, e + I. e + 2. c + 3) (pesar primera y segunda con tercera y cuarta 1
si pi = eq **entonces {** las 4 primeras son auténticas }
{ quitamos las 3 primeras)
(j, *p)* := moneda-falsa(Ai[1 *f* — 2, *f)*
si no { las 2 últimas son auténticas)
{i, p) := moneda-falsa(M[l..n].c. *f* — 2)
fsi
0 m > 6 -> { caso general: hay al menos 7 monedas)
k := m div 2
si impar(Zr) **entonces** *k := k* + 1 **fsi**
k' := k div 2
p pesarte, c + *k'* — 1. *c* + *k', c* + *k* — 1) { dos montones del mismo tamaño }
si *p* = eq **entonces** (buscar entre las restantes }
(i, p) := moneda-falsa(A/[l..nJ, *c* + *k, f)*
si no { buscar entre las pesadas)
(i.p) := moneda-falsa(M[l..n], c, *c* + *k* — 1)

fsi
fcasos
ffun

En los casos básicos hacemos 2 pesadas, y en los recursivos hacemos una sola pesada y nos quedamos con la mitad de monedas (más o menos). Por tanto, el coste del algoritmo está en C-)(log *n).*

Una alternativa es comenzare! algoritmo repartiendo las monedas en tres montones del mismo tamaño (las que sobren se tratan aparte) y realizando dos pesadas. De esta manera se descartan 2/3 de las monedas y, además, se averigua si la moneda falsa pesa más o menos que las auténticas (con esta información, ya no es necesario que haya al menos tres monedas). A continuación se sigue el proceso siempre repartiendo en 3 montones, pero realizando una sola pesada.

El algoritmo resultante es el siguiente:

$\{n \geq 3\}$

fun moneda-falsa2(M[1..n] **de** *moneda)* **dev** *(i :* 1..n, *p : peso)*
k := *n* div 3
p_1 := pesar(1, *k, k*+1, 2**k)*
p_2 := pesar(1, *k,* 2**k*+1, 3**k)*
{cada montón con un tercio de monedas}

casos
p_1=izq ∧ p_2=izq → {la moneda falsa está en el primer tercio y es más pesada}
p := pesada
i := moneda-fa!sa-rec(M[1,.n], 1. *k, p)*
 p_1=izq ∧ p_2=eq → {la moneda falsa está en el segundo tercio y es más ligera}
p := ligera
i := moneda-falsa-rec(M[1..n], *k,* 2**k, p)*
 p_1=eq ∧ p_2=izq → {la moneda falsa está en el tercer tercio y es más ligera}

```
      p := ligera
      i := moneda-falsa-rec(M[1..n], 2*k, 3*k, p)
   [] p1=eq ∧ p2=der → {la moneda falsa está en el tercer
   tercio y es más pesada}
      p := pesada
      i := moneda-falsa-rec(M[1..n], 2*k, 3*k, p)
   [] p1 = der ∧ p2 = eq → {la moneda falsa está en el
   segundo tercio y es más pesada}
      p: pesada
      i:= moneda-falsa-rec(M[1 ..n], k, 2*k, p)
   [] p1=der ∧ p2 = der → {la moneda falsa está en el primer
   tercio y es más ligera}
      p := ligera
      i := moneda-falsa-rec(M[1..n], k, p)
   [] p1 = eq ∧ p2 = eq → {la moneda falsa está entre las
   restantes}
      r := n mod 3
      casos
        r = 0 → error(No hay moneda falsa)
      [] r=1 → { la moneda falsa es la última}
         i := n
         p1:= pesar(n, n, 1, 1) {comparamos con la primera}
         si p1 = izq entonces  p := pesada
         si no p:= ligera
         fsi
      [] r = 2 → {hay 2 candidatas}
         p1 := pesar(n, n, 1, 1) {comparamos primera y última}
         casos
            p1 = izq → i:= n ; p:= pesada
         [] p1 = der → i:= n ; p:= ligera
         [] p1 = eq → {la moneda falsa es la penúltima }
            i := n - 1
            p1 := pesar(n - 1, n - 1. 1, 1) {comparamos
            primera y penúltima}
            si p1 = izq entonces  p := pesada
            si no  p := ligera
            fsi
         fcasos
      fcasos
   fcasos
ffun
```

```
fun moneda-falsa-rec(M[1..n] de moneda, c, f : nat,
p: peso) dev i : 1..n
m : = f - c + 1 {cantidad de monedas}
casos
m = 1 → i := c
```

 m = 2 → {comparamos las dos monedas }
p_1 := pesar(c, c, *f*, *f*)
casos
(pi = izq A *p* = pesada) v (p, = der A p = ligera) —*•
i := *c*
▫ (pi = izq A p = ligera) V (pi = der A p = pesada) —> *i* := *f*
▫ pi = eq —» errorfNo hay moneda falsa)
fcasos
ü *ni* > 2 -> *k* := *ni* div 3 pi := pesarfc, *c* + *k* - 1. *c* + *k*, *c* + 2 * *k* — 1) (cada montón con un tercio de monedas)
casos
(pi = izq A p = pesada) V (pi = der *A p* = ligera) —»
{ la moneda falsa está en el primer tercio)
i := moneda-falsa-rec(M[l..zr], c, *c* + *k* — 1. p)
▫ (pi = izq A p = ligera) v (pj = der A p = pesada) -»
{ la moneda falsa está en el segundo tercio)
i := moneda-falsa-rec(A7[1 *..n]. c* + *k,c* + *2*k* — 1. p)
▫ pi = eq -> (la moneda falsa está entre las restantes }
i := moneda-falsa-rec(A7[l,.n], *c* + 2 * *k*. *f*. p) **fcasos**
fcasos
ffun

El número de pesadas realizadas es $\log_3 n + 1 \in \Theta(\log n)$ (nótese que $\log3(2n) = \log_3 n + \log_3 2$).

11.21. Hanoi

Las ***TORRES DE HANOI*** son 3 varillas donde se pueden insertar *n* discos, cada uno de diferente tamaño. Inicialmente los *n* discos están formando una torre en orden decreciente (de abajo arriba) de tamaño en la primera varilla. El problema consiste en trasladar esta torre de discos a la tercera varilla, pudiendo utilizar la segunda varilla como auxiliar. En cada instante se puede mover un único disco desde una varilla hacia otra, con la restricción de que nunca se puede colocar un disco sobre otro de menor tamaño.

(a) Demostrar que el problema de las Torres de Hanoi con *n* discos no puede resolverse con menos de 2^n - 1 movimientos.

(b) Escribir un algoritmo recursivo que resuelva el problema de las Torres de Hanoi con 2^n - 1 movimientos.

-------------------------Solución:--------------------------

Apartado (a)

Lo demostramos por inducción sobre *n.* Para *n*=1. es necesario al menos un movimiento para llevar el único disco desde la primera varilla hasta la tercera.

En el paso de inducción, cuando se tienen *n*+1 discos en algún momento habrá que trasladar el disco más grande desde la primera varilla hacia la tercera (al menos un movimiento). Para ello es necesario haber quitado previamente los *n* discos superiores de la primera varilla y que estén todos formando una torre en

otra varilla (al menos 2^n -1 movimientos, por hipótesis de inducción); posteriormente, habrá que colocarlos sobre el disco grande en la tercera varilla Así que se realizarán, como mínimo, los siguientes movimientos:

$$2^n - 1 + 1 + 2^n - 1 = 2 * 2^n - 1$$
$$= 2^{n+1} - 1$$

Apartado (b)-------------------------------------

Un algoritmo *recursivo* para resolver las Torres de Hanoi con *n* discos consiste en

- trasladar primero los *n*-1 discos superiores desde el origen hasta la varilla auxiliar;
- a continuación se traslada el disco mayor desde el origen al destino;
- y por último, se vuelven a *trasladar* los *n-1* discos desde la varilla auxiliar hasta el destino.

El procedimiento mover(A, *B : torre)* quita el disco superior de la torre A *y* lo coloca sobre la torre *B.*

```
proc Hanoi(origen, destino, auxiliar : torre, e n : nat)
  si n=1 entonces
    mover(origen, destino)
  si no
    Hanoi(origen, auxiliar, destino, n-1)
    mover(origen, destino)
    Hanoi(auxiliar, destino, origen, n-1)
  fsi
fproc
```

El número de movimientos se obtiene a partir de la siguiente recurrencia:

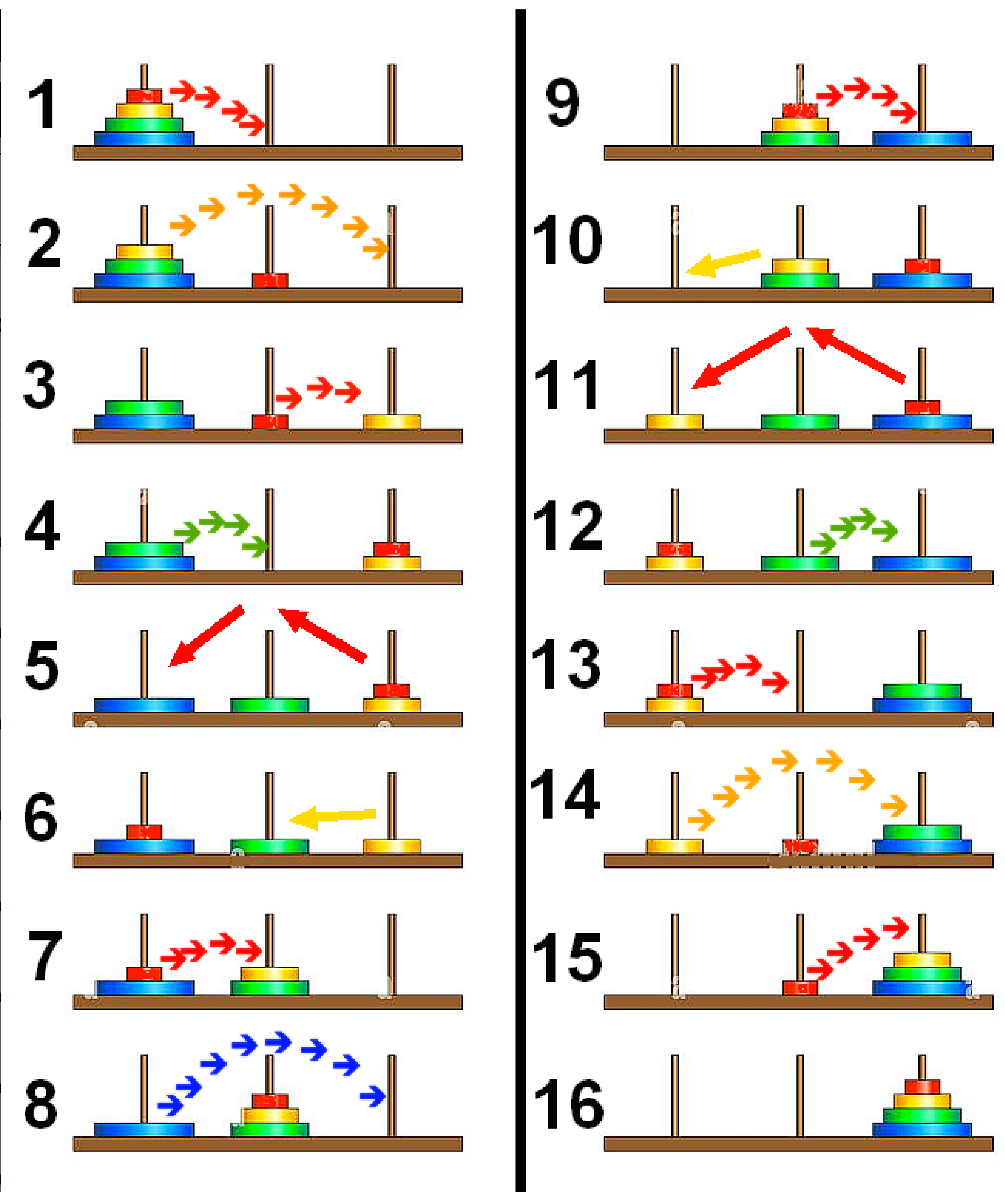
$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1)+1 & n > 1 \end{cases}$$

Resolviendo la misma tenemos:

$$T(n) = 2^i T(n-i) + \sum_{j=0}^{i-1} 2^j = 2^{n-1}T(1) + \sum_{j=0}^{n-2} 2^j$$

$$= \sum_{j=0}^{n-1} 2^j = 2^n - 1$$

Es decir, que según lo demostrado en el primer apartado, este algoritmo es óptimo, pues realiza el mínimo número posible de movimientos.



11.22. Hanoi limitado

Escribir un algoritmo recursivo para resolver el problema de las Torres de Hanoi (Ejercicio 11.21) en el caso de que ***NO* se permitan movimientos entre las torres *origen* y *destino*;** es decir, todos los movimientos tienen que ser desde o hasta la torre ***AUXILIAR***. ¿Cuántos movimientos se realizan?

--------------------------Solución-------------------------

Ahora el movimiento *mover(origen. destino)* del algoritmo recursivo dado en el Ejercicio 11.21 no se puede realizar, por lo que tendremos que desdoblarlo en 2 movimientos: mover(*origen*, *auxiliar)* y mover*(auxiliar, destino).* El algoritmo queda ahora:

```
proc Hanoi-auxiliar(origen, destino,
auxiliar : torre, e n : nat)
 si n=1 entonces
    mover (origen, auxiliar)
    mover(auxiliar, destino)
 si no
    Hanoi-auxiliar(origen, destino, auxiliar, n-1)
    mover(origen, auxiliar)
    Hanoi-auxiliar(destino, origen, auxiliar. n-1)
    mover(auxiliar, destino)
    Hanoi-auxiliar(origen, destino, auxiliar, n-1)
 fsi
fproc
```

Nótese que en las llamadas recursivas cambian las torres *origen* y *destino*, pero la torre utilizada como *auxiliar* es siempre la misma. Esto garantiza que todos los movimientos son correctos.

El número de movimientos se obtiene a partir de la siguiente recurrencia:

$$T(n) = \begin{cases} 2 & n = 1 \\ 3T(n-1) + 2 & n > 1 \end{cases}$$

Resolviendola tenemos:

$$T(n) = 3^i T(n-i) + 2\sum_{j=0}^{i-1} 3^j$$

$$= 3^{n-1}T(1) + 2\sum_{j=0}^{n-2} 3^j = 2\sum_{j=0}^{n-1} 3^j$$

$$= 3^n - 1$$

11.23. Hanoi 4 palos

Escribir un algoritmo recursivo eficiente para resolver el problema de las Torres de **Hanoi** cuando se dispone de **4 varillas**. ¿Cuántos movimientos se realizan?

-----------------------------Solución-------------------------------

Podemos aprovechar que tenemos 2 varillas auxiliares para, en el caso general, en lugar de trasladar recursivamente los *n*-1 discos superiores, trasladar solamente *n*-2 a la primera varilla auxiliar. A continuación trasladaremos el penúltimo disco a la segunda varilla auxiliar, pudiendo entonces trasladar el disco más grande directamente a su destino. Después colocaremos el penúltimo en el destino y. recursivamente, los *n*-2 restantes.

```
proc Hanoi-cuatro1 (origen, destino, aux1, aux2: torre, e n:nat)
casos
    n = 1 → mover(origen, destino)
  ▯ n = 2 →
      mover (origen, aux1)
      mover(origen, destino)
      mover(aux1, destino)
  ▯ n > 2 →
      Hanoi-cuatro1 (origen, aux1, destino, aux2, n-2)
      mover(origen, aux2)
      mover(origen, destino)
      mover (aux2, destino)
      Hanoi-cuatro1 (aux1, destino, origen, aux2, n-2)
```

fcasos
fproc

El número de movimientos se obtiene a partir de la recurrencia

$$T(n) = \begin{cases} 1 & n = 1 \\ 3 & n = 2 \\ 2T(n-2)+3 & n > 2 \end{cases}$$

Resolviendo la misma tenemos:

$$T(n) = 2^i T(n-2i) + 3\sum_{j=0}^{i-1} 2^j$$
$$= 2^{n \operatorname{div} 2}(nmod2 + 3) - 3$$
$$\in O\left(2^{n/2}\right).$$

Se podría pensar en aprovechar mejor la segunda varilla auxiliar para desarrollar un algoritmo de tipo divide y vencerás, trasladando la mitad de los discos en lugar de *n*-2. Pero en las llamadas recursivas hay que tener cuidado para no violar la regla de no poner nunca un disco mayor encima de otro menor. Una posibilidad es utilizar solamente una varilla auxiliar, usando el algoritmo Hanoi del Apartado (b) del Ejercicio 11.21. El algoritmo sería:

proc Hanoi-cuatro2(oríge/i, *destino, aux\, auxs : torre,* **c** *n : nat)*
si *n* **= 1 entonces** *mover(origen, destino)*
si no
Hanoi(oríg<m, *auxi. auX2,* L'i/2J)
Hanoi(oríg<?n, *destino, aux2,* fn/2j)
Hanoi(nuxj. *destino, auxi.* L"/2J)
fsi

fproc

Teniendo en cuenta que el número de movimientos realizados por el algoritmo "clásico" Hanoi para *n* discos es 2" — 1 (véase el Ejercicio 11.21), el número de movimientos para Hanoi-cuatro2 es:

T('n) = { 2 * 2L,,/2J + 2r"/21 - 3 *n* > 1

obteniéndose exactamente el mismo número de movimientos que para Hanoi-cuatrol.

Pero nótese que podemos mejorar este último algoritmo aprovechando que los [n/2J discos superiores son todos menores que los restantes, por lo que para desplazarlos se podrán utilizar todas las varillas.

proc Hanoi-cuatro3(ortg<?n. *destino.* <*iux\. 011x2 : torre,* **e** *n* : nat)
si *n* = 1 **entonces** *moveriorigen, destino)*
si no
Hanoi-cuatroSforíge/i, *anxi. destino. 011x2.* Ln/2J)
Hanoiforigen. *destino, aux2,* f/t/2])
Hanoi-cuatro3(nn.ri, *destino, origen, 011x2,* [n/2J)
fsi
fproc

Entonces el número de movimientos se obtiene a partir de la recurrencia:
f('!) = | 27’((n/2J)+2r"/21 - 1 *n* > 1
y suponiendo *n* potencia de 2, obtenemos:

i *i* — I
T(n) = 2'T(n/2') + £j2J"l2',/2J) - ^2'
;=l j=0
log/I
= 1 + - ¿ *2j2n,1*'
~ ;=i
loen
= 1 + 1 £2'+'-/<
- J=1

En definitiva, se tiene *Tin)* > 2"/2 e Q(2"/2). Por otro lado, puesto que para todo *j* con 1 < *j* < logn se cumple que *j* + n/2J < (J — 1) + n/2J-1, se tiene que
! <>/2+1
Tin) < 1 + -(2n + 2')
i=0
= 1/2 + H + 2"/2+I

e O(2"/2).

Resumiendo, *Tin)* e 0(2"/2).

Podemos mejorar aún más este algoritmo? En nuestra última versión hemos combinado el método clásico (para tres varillas) con el método "mejorado” (para cuatro varillas), pero el reparto de discos se ha hecho de forma equitativa. Si aprovechamos las cuatro varillas para desplazar más discos, ¿lograremos reducir el número de movimientos? La respuesta es afirmativa. Consideremos una modificación del algoritmo anterior, donde el reparto es *n* — discos para la llamada recursiva (cuatro varillas) y [v'nj discos para el método clásico (tres varillas).

proc Hanoi-cuatro4(origen, *destino, auxi, aiiX2* ■ *torre, en : nat)*

si *n* = 1 **entonces** *mover (origen, destino)*

si no

A- ;= Ls/”J

Hanoi-cuatro4(orígen, «tt.vi. *destino, aiiX2.n — k)*

Hanoi(origen, *destino, 011x2. k)*

Hanoi-cuatro4(au.V|, *destino, origen, 011x2. n — k)*

fsi fproc

Entonces el número de movimientos corresponde a la recurrencia:

11 *n* = 1

7 (/,) = | 2*Tin* - L Al) + - l *n* > 1

Vamos a demostrar por *inducción*

constructiva que *Tin)* edemostrando la existencia de cons
tantes reales positivas *a* y *b* tales que *T(n)* < $a2^{b}$*^".* En el caso básico. *T(* 1) = 1 < $a2^{b}$ para *a* > I y *b* > 0.

En el paso de inducción, supongamos que *T(m)* < $a2^{b}$*'^"* para *m* < *n.* Entonces tenemos

T(n) = *2T(n* - LV»]) + 2^ - 1

<>•'■ *2(a2^h'/"~^,h* + 2^J - 1

< +2⁶^J *{ b* > 1)

< *2a2^{l,}'^"2~^b'^" 2^{by}'"~* + 2*'/Z"

— 2*^" + 2a2í"/"2_/'('^"'V/"- L^"J)

_ *2^b'/"(l+*

< 2^{i,}'/"(l +a2'-*/²) *{* - *y/n* - | *)*

< 2^{fr}'/"(l+a2'-⁴/²) {si/>>4)

= 2^b'/"(l+«/2)

< *a2^b'/"* (si *a* > 2)

< Por tanto, la propiedad es cierta tomando *a* > 2 y *b* > *4.*

Capítulo 12

12. MÉTODO VORAZ

ESQUEMA GENERAL

Las características generales de los algoritmos voraces son:

- Para construir la solución se dispone de un conjunto de candidatos. A medida que avanza el algoritmo se van formando 2 conjuntos: el conjunto de candidatos considerados y **seleccionados** (formarán parte de la solución), y el conjunto de candidatos considerados y **rechazados** definitivamente.
- Existe una ***función de selección*** que indica cuál es el candidato más prometedor de entre los aún no considerados.
- Existe un ***test de factibilidad*** que comprueba si un candidato es compatible con la solución parcial construida hasta el momento: esto es. si existe una solución incluyendo dicha solución parcial y el citado candidato.
- Existe un ***test de solución*** que determina si una solución parcial forma una solución "completa".
- Con frecuencia, se trata de problemas de *optimización* (es decir, se tiene que obtener una solución óptima según una *función objetivo* que asocia un

valor a cada solución).

La ***función objetivo*** se tiene en cuenta en la definición de la función de selección, pero no influye en los tests de factibilidad y de solución, que son independientes de la optimalidad de la solución considerada.

En general, los algoritmos voraces intentan optimizar la función objetivo, considerando en cada paso el candidato devuelto por la función de selección y añadiéndolo al conjunto de seleccionados, con lo que pasa a formar parte de la solución, siempre que se siga pasando el test de factibilidad, o al de rechazados. en caso contrario. Lo fundamental es que cada candidato se trata una sola vez. es decir, no hay posibilidad de replantearse una decisión tomada. De esta forma, al considerar una única posibilidad, los algoritmos son sencillos y eficientes. En cambio, la demostración de su corrección suele ser complicada. Para demostrar que la solución de un algoritmo voraz es óptima, suele aplicarse el método de *reducción de diferencias,* consistente en comparar una solución óptima con la solución obtenida por el algoritmo voraz. Si ambas soluciones no son iguales, se va transformando la solución óptima de punida de forma que continúe siendo óptima, pero siendo más parecida a la del algoritmo voraz. Cuando las soluciones tienen un número finito de etapas, repitiendo el proceso un número finito de veces se llega a que ambas coinciden y,

por tanto, la solución del algoritmo voraz también es óptima.

El esquema general que siguen los algoritmos voraces es el siguiente:

```
fun voraz(datos : conjunto) dev S : conjunto
var candidatos : conjunto
S := ∅   {en S se va construyendo la solución}
candidatos := datos
mientras candidatos ≠ ∅ ∧ ¬ solución (S) hacer
    x := seleccionar(candidatos)
    candidatos := candidatos - {x}
    si factible(S U {x}) entonces
        S := S U{x}
    fsi
fmientras
ffun
```

EJERCICIOS RESUELTOS VORACES

12.1. Tareas

Un sistema da servicio a *n* tareas, cada una con un tiempo de ejecución ti para *i* entre 1 y *n*. Se desea minimizar el tiempo medio de estancia de una tarea en el sistema, esto es, el tiempo transcurrido desde el comienzo de todo el proceso hasta que la tarea termina de ejecutarse.

Resolver el problema cuando

(a) se dispone de un único procesador;

(b) se tienen s procesadores idénticos.

--------------------------Solución:------------------------

Apartado (a)

El problema consiste en minimizar

$$T = \frac{1}{n}\sum_{i=1}^{n} T_i$$

donde Ti es el tiempo en el sistema de la tarea *i*, y como *n* está fijo, esto es equivalente a minimizar el tiempo total

$$\sum_{i=1}^{n} T_i$$

Para la tarea i, Ti es la suma de su tiempo de ejecución r, más todos los tiempos de las tareas que se ejecutan antes que ella. Veremos que la planificación óptima se consigue cuando se atiende a las tareas por orden creciente de tiempo de ejecución. En efecto, sea I = $i_1,\ldots,i_n$ una permutación cualquiera de los enteros del 1 al *n*, que define un posible orden de realizar las tareas. El tiempo total correspondiente a esta permutación será

$$T(I) = t_{i_1} + \left(t_{i_1} + t_{i_2}\right) + \left(t_{i_1} + t_{i_2} + t_{i_3}\right) + \cdots + \left(t_{i_1} +\right.$$
$$= nt_{i_1} + (n-1)t_{i_2} + \cdots + 2t_{i_{n-1}} + t_{i_n}$$
$$= \sum_{k=1}^{n} (n-k+1)t_{i_k}.$$

Por tanto, la contribución del tiempo de cada tarea al tiempo total depende del número de tareas que la siguen.

Supongamos que la permutación I no organiza las tareas por orden creciente de tiempos. Entonces, existen posiciones *a* y *b* tales que $a < b$ y $ti_a > ti_h$, de modo que podemos obtener una nueva permutación I', intercambiando *ia* con ib en *I:*

$I' =$ $i_1,\ldots,i_{a-1}$, i_b, i_{a+1}, i_{b-1},i_a, $i_{b+1},\ldots$, i_n.

El tiempo total correspondiente a I' será entonces

$$T(I') = (n-a+1)t_{i_b} + (n-b+1)t_{i_a} + \sum_{\substack{k=1 \\ k\neq a,b}}^{n} (n-k+1)t_{i_k}.$$

Si restamos ambos tiempos obtenernos:

$$T(I) - T(I') = (n-a+1)\left(t_{i_a} - t_{i_b}\right) + (n-b+1)\left(t_{i_b} - t_{i_a}\right)$$
$$= \left(t_{i_a} - t_{i_b}\right)(n-a+1-(n-b+1))$$
$$= \left(t_{i_a} - t_{i_b}\right)(b-a) > 0.$$

Por tanto, la permutación I' proporciona un tiempo mejor. Es decir, cualquier permutación que no siga el orden indicado se puede mejorar, deduciéndose que toda permutación que coloca las tareas por orden creciente de tiempos es óptima.

Así pues, el algoritmo pedido consiste simplemente en ordenar las tareas según el orden establecido en la estrategia.

En realidad, la optimalidad de la estrategia anterior se obtiene como caso particular del siguiente resultado.

Lema:

Dados *n* valores positivos ordenados $v_1 \leq ... \leq v_n$, y n coeficientes positivos $c_1,...,c_n$ se cumple que la suma 5 = *ci*$_t$*vii* se minimiza cuando los coeficientes están ordenados de forma decreciente: $c_{i1} > ... \geq c_{in}$.

Demostración:

Supongamos que no están ordenados de esa forma, entonces existen posiciones *a* y *b* tales que $a < b$ y $c_{ia} < c_{ib}$. Por hipótesis se tiene también $v_a \leq v_b$. En el caso de que v_a = vb. está claro que el orden entre *a y b* no importa, ya que podemos intercambiar los valores cia y cib sin modificar el valor de la suma. Consideremos entonces el caso en el que $v_a < vb$. Si calculamos la suma S' obtenida al intercambiar en *S* los coeficientes ci_a *y* ci_b obtenemos

$$S' = c_{i_b} v_a + c_{i_a} v_b + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} c_{i_k} v_k .$$

Para ver que S' es mejor que S, es decir S' < S, las restamos obteniendo

$$S - S' = \sum_{k=1}^{n} c_{i_k} v_k - \left(c_{i_b} v_a + c_{i_a} v_b + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} c_{i_k} v_k \right)$$
$$= c_{i_a} v_a + c_{i_b} v_b - c_{i_b} v_a - c_{i_a} v_b$$
$$= c_{i_a}(v_a - v_b) - c_{i_b}(v_a - v_b)$$
$$= \left(c_{i_a} - c_{i_b}\right)(v_a - v_b) > 0.$$

Lo que, unido al razonamiento correspondiente al caso en el que teníamos $v_a = v_b$, nos conduce a la optimalidad de la solución planteada.

Para aplicar este lema al problema de la planificación de tareas con un procesador único basta tomar como valores los tiempos de ejecución de las tareas y como coeficientes el conjunto de enteros {1,…,n}.

Apartado (b)------------------------------

Supongamos ahora que disponemos de s procesadores idénticos. Observamos que si a un procesador *p* se le asignan m tareas y a otro procesador *p′* se le asignan *m* +*l* tareas, con l > 1, entonces, el tiempo de servicio de la primera tarea asignada a *p′* se multiplica por un factor *m* +*l.* mientras que si se trasladara al primer puesto en *p* (desplazando las m tareas) tendría un factor $m + 1 < m + l$ sin que el resto del tiempo computado por ambos procesadores se viera afectado porque para las restantes tareas no cambiaría el número de tareas que las siguen. De esta forma, el tiempo total decrecería estrictamente. Esto significa que cualquier planificación óptima debe mantener *equilibrada* la carga de todos los procesadores, de forma que la diferencia en el número de tareas asignadas sea a lo sumo uno, es decir, la carga de cada procesador será n div s o n div s + 1. Además, como los procesadores son idénticos, podemos intercambiarlos entre sí, de forma que los que tengan más tareas sean los primeros. De esta forma, para una planificación óptima, los coeficientes que afectarán a los tiempos de servicio de las *n* tareas son

$$\{\overbrace{1,\ldots,1}^{s},\overbrace{2,\ldots,2}^{s},\ldots,\overbrace{n\text{ div }s,\ldots,n\text{ div }s}^{s},\overbrace{(n\text{ div }s)+1,\ldots,(n\text{ div }s)+1}^{n\text{ mod }s},$$

porque hay s últimas, s penúltimas, etc.

De nuevo consideramos las tareas ordenadas por orden creciente de tiempo

de servicio (r, < *ti* < ... < í„) para una planificación *P* que reparte las tareas entre los procesadores como si de naipes se tratara:

$$p_1 \longmapsto t_1, t_{s+1}, t_{2s+1}, \ldots$$
$$p_2 \longmapsto t_2, t_{s+2}, t_{2s+2}, \ldots$$
$$p_s \longmapsto t_s, t_{2s}, t_{3s}, \ldots$$

De esta forma, al procesador *pj* le corresponderá la secuencia de tareas $t_1^j, \ldots, t_n^j$ donde $t_k^j = t_{(k-1)s+j}$ para *k* entre 1 y n_j, y

$$n_j = \begin{cases} n \operatorname{div} s + 1 & \text{si } j \leq (n\, mod\, s) \\ n \operatorname{div} s & \text{si } j > (n\, mod\, s) \end{cases}$$

El tiempo asociado a esta planificación es

$$T(P) = \sum_{j=1}^{s} \sum_{k=1}^{n_j} (n_j - k + 1) t_{(k-1)s+j},$$

y el lema anterior garantiza su optimalidad.

12.2. Programas en disco

Consideremos *n* programas *P1,…,Pn* que debemos almacenar en un disco. El programa *Pi* requiere *si* megabytes de espacio de disco, y la capacidad del disco es *D* megabytes.

(a) Se desea maximizar el *número de programas* almacenados en el disco. Demostrar lo siguiente o dar un contraejemplo: podemos utilizar un algoritmo voraz que seleccione los programas por orden creciente de *si.*

(b) Se desea maximizar el *espacio* utilizado del disco. Demostrar lo siguiente o dar un contraejemplo: se puede utilizar un algoritmo voraz que seleccione los programas por orden decreciente de *si.*

------------------------Solución:------------------------

Supondremos que la capacidad del disco es insuficiente para contener todos los programas, es decir, que $D < \sum_{i=1}^{n} s_i$; en caso contrario, la solución en ambos casos es trivial: almacenar los *n* programas en el disco.

Apartado (a)---

La estrategia propuesta considera los programas de menor a mayor tamaño, y para cada programa realiza la siguiente operación: si el programa cabe en el espacio de disco todavía no utilizado, graba el programa y pasa al siguiente; y si el programa no cabe, lo descarta y termina, pues los siguientes tampoco van a caber.

Suponiendo que los programas están ya ordenados de menor a mayor tamaño, vamos a demostrar que la estrategia conduce siempre a una solución óptima, con el mayor número posible de programas grabados. Lo haremos mediante el método de *reducción de diferencias* (Sección 12.1), comparando la solución devuelta por la estrategia voraz. *X.* con otra solución óptima. *Y.*

Para ello representamos una solución cualquiera Z mediante su función característica, es decir, con una tupla $(z_1,...,z_n)$ donde $z_i=0$ indica que el programa *Pi* no se coge, mientras que $z_i=1$ indica que *Pi* forma parte de la solución. Para ella el número de programas seleccionados viene dado por $\sum_{i=1}^{n} z_i$.

Supongamos que la estrategia voraz propuesta escoge los *k* primeros programas (con $1 \leq k \leq n$) y el *(k+1)*-ésimo ya no cabe, por lo que se descarta junto con los restantes. Entonces, en la solución *X* se tiene $\forall i{:}1 \leq i \leq k : x_i=1$ y $\forall i{:} k<i \leq n : x_i=0$. El número de programas correspondiente es obviamente *k.*

Empezando a comparar (de izquierda a derecha) con *Y,* sea $j \geq 1$ la primera posición donde $x_j \neq y_j$. En primer lugar, nótese que $j \leq k$ pues en caso contrario la solución óptima incluiría todos los programas escogidos por la estrategia voraz y alguno más, lo cual es contradictorio con el hecho de que los

programas se rechazan porque no caben.

Se tiene, por tanto, la siguiente situación:

1	1	…	1	…	1	…	0	…	0
x_1	x_2	…	x_j	…	x_k	…	x_l	…	x_n
$=$	$=$	…	$\neq$				$\neq$		
y_1	y_2	…	y_j	…	y_k	…	y_l	…	y_n

Entonces, si $y_j \neq x_j=1$, tenemos $y_j=0$, y de aquí se sigue que $\sum_{i=1}^{j} y_i = j-1 < j = \sum_{i=1}^{j} x_i$. Ahora bien, como suponemos que *Y* es óptima, $\sum_{i=1}^{n} y_i \geq \sum_{i=1}^{n} x_i = k$. Por tanto, existe *l>k≥j* tal que yl=1, es decir, tiene que haber un programa posterior para compensar el que no se ha cogido antes. Ahora bien, por la ordenación de los programas sabemos que *sj≤sl, es* decir, que si P_l cabe en el disco, podemos poner en su lugar P_j sin sobrepasar la capacidad total. Realizando este cambio en la solución óptima *Y* obtenemos otra solución *Y'* en la cual $y'_j=1=x_j$, $y'_l=0$ *y* para el resto $y'_i=y_i$. Esta nueva solución es más parecida a *X, y* tiene el mismo número de programas que Y por lo que sigue siendo óptima. Repitiendo este proceso, podemos ir igualando los programas en la solución óptima a los de la solución voraz *X,* hasta alcanzar la posición *k.*

Suponiendo que los programas ya están en orden creciente de tamaño, el algoritmo que implementa esta estrategia es el siguiente:

{S[1]≤...<S[n] ∧ $\sum_{i=1}^{n} S[i]$ >D }

```
fun programas-en-disco(S[1..n] de real+, D : real+) dev sol[1..n] de 0..1

sol[1..n] := [0]
ocupado := 0 ;
i:= 1
mientras ocupado + S[i] ≤ D hacer
    sol[i] := 1
    ocupado := ocupado + S[i]
    i:= i + 1
fmientras
ffun
```

Como hemos asumido que $D < \sum_{i=1}^{n} S[i]$, no es necesario comprobar en el bucle del algoritmo que *i* no sobrepasa el límite de *n*.

Apartado (b)---

Veamos con un contraejemplo que la estrategia que considera los programas en orden decreciente de tamaño no siempre encuentra la solución óptima al problema de maximizar el espacio utilizado del disco.

Si tenemos 3 programas con los tamaños siguientes:

s_1	s_2	s_3
30	25	20

y un disco de tamaño *D* = 45. la estrategia consideraría el programa 1, que entraría en el disco, tras lo cual no podría incluir ninguno más. En cambio, la solución óptima sería incluir los programas 2 y 3.

Tampoco sirve considerar los programas en orden creciente. Para los

programas de tamaños

s_1	s_2	s_3
10	15	20

y un disco de tamaño D = 30, la nueva estrategia incluiría los programas 1 y 2, con un espacio total de 25. mientras que la solución óptima que graba los programas 1 y 3 llena completamente el disco.

Este problema puede resolverse utilizando programación dinámica (Capítulo 13). vuelta atrás (Capítulo 14) o ramificación y poda (Capítulo 15).

12.3. Intervalos

Dado un conjunto $\{r_1,\dots,r_n\}$ de puntos de la recta real, determinar el menor conjunto de intervalos cerrados de longitud 1 que contenga a todos los puntos dados.

--------------------------Solución:------------------------

Suponemos que tenemos los puntos (números reales) del conjunto ordenados en un vector R[1..n]. La estrategia voraz consiste en tomar el intervalo unitario que comience en el primer punto considerado y ver qué puntos cubre, para repetir el proceso a partir del primer punto todavía no cubierto, hasta que se hayan cubierto todos los puntos.

El algoritmo es el siguiente:

```
tipos
  intervalo = vector[1..2] de real
ftipos
{R[1][1] < … < R[n] }
fun cubrir(R[1..n] de real) dev C : conjunto[intervalo]
var I : intervalo

C := cjto-vacio() ;
j := 1
mientras j ≤ n hacer
  I[1] := R[j];
  I[2] := R[j]+ 1
  añadir(I, C)
  mientras j≤n ∧c R[j]≤I[2] hacer {saltar puntos ya cubiertos}
    j := j + 1
  fmientras
fmientras
ffun
```

Una vez ordenados los puntos, el coste del bucle voraz es lineal respecto a la cantidad de puntos dados.

<u>Demostramos por reducción de diferencias que el algoritmo es óptimo</u>: sea $X = [x_1,\ldots,x_l)$ el conjunto (ordenado) de intervalos obtenido por el algoritmo voraz, e $Y = \{y_1,\ldots,y_k\}$ una solución óptima (por tanto, $k \leq l$) también ordenada. Comparamos X e Y de izquierda a derecha hasta la primera posición i donde sean diferentes. El intervalo $y_i = [c_y,f_y]$ no puede empezar después que $x_i = [c_x,f_x]$ porque si no el punto colocado en el extremo izquierdo (c_v) no seria cubierto por Y. Así $c_y < c_x$ y podemos cambiar y_i por x_i para obtener otra solución, también óptima, que se parezca más a X. Es solución porque sigue cubriendo todos los puntos, ya que en el intervalo $[c_y,c_x)$ no hay puntos que no cubran los intervalos $y_1,\ldots, y_{i-1}$ (si no el algoritmo voraz no hubiera empezado el intervalo en c_x) y por la derecha hemos retrasado el final del intervalo ($f_y<f_x$) por lo que no se ha dejado ningún punto sin cubrir. Además, sigue siendo óptima porque el número de intervalos no ha cambiado. Repitiendo este proceso llegamos a una solución óptima $Y' = \{x_1,\ldots,x_k\}$, que ha de verificar $k=l$ porque si no, los puntos correspondientes a los comienzos de los intervalos $x_{k+1},\ldots,x_l$ quedarían sin cubrir.

12.4. Una cinta magnética contiene *n* programas de longitudes $l_1,\ldots l_n$. Se supone que tanto la densidad de información en la cinta como la velocidad de lectura son constantes, y que. tras cada búsqueda seguida de la lectura de un programa, la cinta es automáticamente rebobinada. Se conoce la tasa de utilización de cada programa: esto es, se sabe que del número total de peticiones, un porcentaje *pi* corresponde al programa *i* ($1 \leq i \leq n$) con $\sum_{i=1}^{n} p_i = 1$.

El objetivo es minimizar el tiempo medio de carga, el cual es proporcional a

$$T = \sum_{j=1}^{n} \left(p_{i_j} \sum_{k=1}^{j} l_{i_k} \right)$$

cuando los programas están almacenados en el orden $i_1,\ldots,i_n$.

(a) Demostrar mediante un contraejemplo que la secuencia en orden creciente de li, no es necesariamente óptima.

(b) Demostrar asimismo mediante un contraejemplo que la secuencia en orden de *pi* decreciente no es necesariamente óptima.

(c) Demostrar que la secuencia ordenada en forma decreciente de p_i/l_i minimiza el tiempo medio de carga.

-------------------------Solución:------------------------

Apartado (a)

El siguiente contraejemplo sirve para demostrar que la estrategia que considera los programas en orden creciente de longitudes no es

necesariamente óptima:

i	1	2	3
li	2	5	8
pi	0,2	0,1	0,7

La estrategia seleccionaría las tareas en el orden dado 1,2,3 con un tiempo medio

T = 0,2 *2+ 0,1 * (2+5)+ 0,7* (2+5 + 8) = 11,6.

En cambio, la solución óptima se consigue colocando los programas en el orden 1,3,2. con un tiempo medio

Topt = 0,2 * 2 + 0,7 * (2+8) + 0,1 * (2+8+5) = 8,9.

Apartado (b)-----------------------------

El mismo contraejemplo nos sirve para probar que la estrategia que considera los programas por orden decreciente de probabilidades tampoco es necesariamente óptima. La estrategia seleccionaría los programas en el orden 3. 1.2. con un tiempo medio

T = 0,7 * 8 + 0,2 * (8 + 2) + 0,1 * (8 + 2 + 5) = 9,1

peor que el óptimo (T_{opt} = 8,9).

Apartado (c)---------------------------

Vamos a demostrar que cualquier solución que no siga la estrategia del enunciado se puede mejorar *y* que, por tanto, la solución que sigue la estrategia es óptima.

Consideremos una solución que *no* tiene los programas ordenados en la forma

$$\frac{p_1}{l_1} \geq \frac{p_2}{l_2} \geq \cdots \geq \frac{p_n}{l_n}$$

donde para simplificar las fórmulas que vienen a continuación vamos a suponer que pi *y li* se refieren al porcentaje y longitud del programa que en la solución aparece en i-ésimo lugar.

Entonces existe una posición *a* tal que los programas *a* y *a*+1 están mal colocados con respecto a la estrategia, es decir

$$\frac{p_a}{l_a} < \frac{p_{a+1}}{l_{a+1}}$$

o equivalentemente $p_a l_{a+1} < p_{a+1} l_a$

El tiempo medio para esta solución es

$$\begin{aligned} T = {} & p_1 l_1 + \\ & p_2(l_1 + l_2) + \\ & \ldots \\ & p_{a-1}(l_1 + \cdots + l_{a-1}) + \\ & p_a(l_1 + \cdots + l_{a-1} + l_a) + \\ & p_{a+1}(l_1 + \cdots + l_{a-1} + l_a + l_{a+1}) + \\ & p_{a+2}(l_1 + \cdots + l_{a-1} + l_a + l_{a+1} + l_{a+2}) + \\ & \ldots \\ & p_n(l_1 + \cdots + l_n). \end{aligned}$$

Consideremos ahora la solución obtenida al intercambiar los programas *a y a*+1, y veamos que es (estrictamente) mejor. Su tiempo medio es

$$
\begin{aligned}
T' = p_1 l_1 + & \\
& p_2(l_1 + l_2) + \\
& \dots \\
& p_{a-1}(l_1 + \cdots + l_{a-1}) + \\
& p_{a+1}(l_1 + \cdots + l_{a-1} + l_{a+1}) + \\
& p_a(l_1 + \cdots + l_{a-1} + l_{a+1} + l_a) + \\
& p_{a+2}(l_1 + \cdots + l_{a-1} + l_{a+1} + l_a + l_{a+2}) + \\
& \dots \\
& p_n(l_1 + \cdots + l_n)
\end{aligned}
$$

Calculando la diferencia T - T' vemos que $T' < T$ y por tanto T no es óptimo: como hasta la posición a-1 y a partir de la posición a+2 los sumandos coinciden en T y T', la diferencia queda

$$
\begin{aligned}
T - T' &= p_a(l_1 + \cdots + l_{a-1} + l_a) - p_{a+1}(l_1 + \cdots + l_{a-1} + l_{a+1}) + \\
&\quad p_{a+1}(l_1 + \cdots + l_{a-1} + l_a + l_{a+1}) - p_a(l_1 + \cdots + l_{a-1} + l_{a+1} + l_a) \\
&= p_a(l_1 + \cdots + l_{a-1} + l_a) - p_a(l_1 + \cdots + l_{a-1} + l_{a+1} + l_a) + \\
&\quad p_{a+1}(l_1 + \cdots + l_{a-1} + l_a + l_{a+1}) - p_{a+1}(l_1 + \cdots + l_{a-1} + l_{a+1}) \\
&= -p_a l_{a+1} + p_{a+1} l_a > 0
\end{aligned}
$$

dado que $p_a l_{a+1} < p_{a+1} l_a$. Y de aquí se sigue $T' < T$ como queríamos.

12.5. Cuando Alí-Babá consigue por fin entrar en la Cueva de los 40 Ladrones encuentra allí gran cantidad de objetos muy valiosos. A pesar de su pobreza, Alí-Babá conoce bien el peso y valor de cada uno de los objetos en la cueva. Debido a los peligros que tiene que afrontar en su camino de vuelta, solo puede llevar consigo las riquezas que quepan en su pequeña mochila, que soporta un peso máximo conocido.

(a) Suponiendo que los objetos son **fraccionables**, determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

(b) ¿Sigue siendo óptima la estrategia utilizada en el apartado anterior para elegir los objetos en el caso de objetos **NO** **fraccionables**?

(c) Supongamos ahora que en la cueva hay una cantidad determinada de riquezas de diferentes clases. Cada riqueza de una clase es fraccionable y tiene cierto peso y cierto valor conocidos. Determinar qué cantidad de riquezas de cada clase debe coger Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

(d) ¿Qué ocurre en el apartado anterior si se considera que para cada clase hay una cantidad inagotable?

------------------------Solución:-----------------------

Apartado (a)

Supongamos que en la cueva hay *n* objetos, cada uno con un peso $p_i>0$ y un valor $v_i>0$ (con *i* entre 1 y n), y consideremos que la mochila soporta un peso total máximo $M>0$. El problema consiste en maximizar

$$\sum_{i=1}^{n} x_i v_i$$

con la restricción

$$\sum_{i=1}^{n} x_i p_i \leq M$$

donde xi es la fracción del objeto *i* tomada, $0 \leq x_i \leq 1$.

Ya que el caso en el que todos los objetos caben juntos en la mochila no tiene mucho interés, consideraremos el caso en el que $\sum_{i=1}^{n} p_i > M$. En este caso, es claro que la solución óptima deberá llenar la mochila por completo. Es decir, se deberá cumplir $\sum_{i=1}^{n} x_i p_i = M$.

Teniendo en cuenta que elegimos fracciones de objetos, y que si un objeto se considera "bueno’. cuanto más tomemos de él mejor, parece obvio que el algoritmo debería ir considerando los objetos, en el orden adecuado, mientras haya espacio en la mochila, y tomar el objeto entero, si cabe, y si no, la fracción de objeto que complete la mochila, en la siguiente forma:

fun mochila-v1 (*P*[1..n], *V*[1..n] **de** *real*$^+$,

```
M: real+) dev sol[1..n] de real

sol[1..n] := [0]
peso := 0
mientras peso < M hacer
   i := el mejor objeto restante
   si peso + P[i] ≤ M entonces
      sol[i] := 1
      peso := peso + P [i]
   si no
      sol[i] := (M - peso)/P[i]
      peso := M
   fsi
fmientras
ffun
```

Queda por determinar la función de selección, es decir, cómo elegimos *el mejor objeto restante.* En principio, tenemos varias posibilidades: podemos seleccionar, cada vez, el objeto más valioso de entre los restantes y así incrementaremos el valor de la carga lo más rápidamente posible; o bien, podemos seleccionar el objeto más ligero, con vistas a aumentar lo más lentamente posible el peso total; o bien podemos evitar ambos extremos, seleccionando el objeto cuyo valor por unidad de peso, *vi/pi,* sea el mayor posible.

Se puede comprobar fácilmente que las 2 primeras estrategias no siempre conducen a una solución óptima. Demostraremos, en cambio, que la tercera posibilidad (seleccionar los objetos por orden decreciente de v_i/p_i) lleva siempre a una solución óptima.

Supongamos que los objetos están

ordenados en forma decreciente respecto a su valor por unidad de peso;

$$\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \cdots \geq \frac{v_n}{p_n}$$

y sea $X= (x_1,\ldots,x_n)$ la solución construida por el algoritmo voraz.

Si todos los x_i son 1, la solución es claramente óptima.

En caso contrario, sea j el menor índice tal que $x_j<1$. Debido a cómo funciona el algoritmo, sabemos que

$x_i=1$ si $i<j$

$x_i=0$ si $i>j$.

Llamemos $V(X)$ al valor total de la solución X, es decir,

$$V(X) = \sum_{i=1}^{n} x_i v_i$$

Vamos a comparar la solución X con una solución Y cualquiera, para llegar a ver que $V(X) \geq V(Y)$; más concretamente, $V(X) - V(Y) \geq 0$ de lo que se deduce que X es óptima.

$$V(X) - V(Y) = \sum_{i=1}^{n} (x_i - y_i)v_i$$

$$= \sum_{i=1}^{n} (x_i - y_i)p_i \frac{v_i}{p_i}$$

Cuando $i<j$, sabemos que $x_i=1$, y por tanto, $x_i-y_i \geq 0$; mientras que, por la forma en que están ordenados los objetos, $\frac{v_i}{p_i} \geq \frac{v_j}{p_j}$.

En cambio, cuando $i>j$, sabemos que $x_i=0$ y en ese caso $x_i-y_i \leq 0$; mientras que

$\frac{v_i}{p_i} \leq \frac{v_j}{p_j}$.

Por último, cuando *i*=*j* es obvio que $\frac{v_i}{p_i} = \frac{v_j}{p_j}$. Así pues, para todo *i*, 1≤*i*≤n, se tiene que

$$(x_i - y_i)\frac{v_i}{p_i} \geq (x_i - y_i)\frac{v_j}{p_j}$$

y por tanto,

$$V(X) - V(Y) \geq \sum_{i=1}^{n} (x_i - y_i)p_i \frac{v_j}{p_j}$$

$$= \frac{v_j}{p_j}\sum_{i=1}^{n} (x_i - y_i)p_i.$$

Ahora bien, $\frac{v_j}{p_j} > 0$ ya que todos los valores y pesos son positivos.

Por otra parte, se cumple

$$\sum_{i=1}^{n} x_i p_i = M \wedge \sum_{i=1}^{n} y_i p_i \leq M.$$

por lo que finalmente

$$V(X) - V(Y) \geq \sum_{i=1}^{n} (x_i - y_i)p_i$$

$$= \sum_{i=1}^{n} x_i p_i - \sum_{i=1}^{n} y_i p_i \geq 0.$$

El algoritmo que implementa esta estrategia es el siguiente:

fun mochila-v2(*P*[1..n], *V*[1..n] **de** *real*⁺, *M*: *real*⁺) **dev** *sol*[1..n] **de** *real*
var *D*[1..n] **de** *real*⁺, I[1..n] **de** 1..n {para ordenar los índices}

para *i* = 1 **hasta** *n* **hacer**
D[i]:= *P*[i]/*V*[i]

```
fpara

I:= ordenarIndices(D);
peso:= 0;
i:= 1;
mientras i≤n ∧c peso + P[I[i]≤M hacer
   {podemos coger el objeto I[i] entero}
   sol[I[i]:= 1;
   peso:= peso + P[I[i]];
   i:= i+1
fmientras

si i≤n entonces {partir el objeto I[i]}
   sol[I[i]:= (M-peso)/P[I[i]]
   para j=i+1 hasta n hacer
      sol[I[j]]:= 0
   fpara
fsi
ffun
```

Para no perder la relación entre los objetos en los vectores *P y* V y los valores correspondientes en el vector *D* sobre el que se calcula el valor por unidad de peso de cada objeto, *D* no se ordena directamente sino utilizando la función ordenarIndices (véase el Ejercicio 11.6) que, a partir del vector *D,* obtiene el vector de índices I, donde I[i] indica el número del objeto que debe ocupar el í-ésimo lugar en la ordenación deseada.

El coste del bucle voraz es lineal respecto al número de objetos, por lo que el coste total del algoritmo es del mismo orden que la ordenación de los objetos.

Apartado (b)----------------------

Si los objetos no son fraccionables. la estrategia voraz anterior no siempre consigue una solución óptima.

Consideremos el siguiente contraejemplo con tan solo dos objetos: un primer objeto con un peso de 1 unidad y un valor de 1 unidad, y un segundo con un peso de 10 unidades y un valor de 5 unidades. Si la mochila admite un peso máximo de 10 unidades, el algoritmo voraz escogería en primer lugar el primer objeto: entonces el segundo objeto ya no cabe, y el valor conseguido es 1. Mientras que si se escoge el segundo objeto, se consigue un valor de 5.

En el Capítulo 13, dedicado a la técnica de programación dinámica, se verá cómo resolver de forma correcta este problema (véase el Ejercicio 13.2). Asimismo, en los Capítulos 14 y 15, ofrecemos soluciones utilizando las técnicas de vuelta atrás y ramificación y poda para la exploración de espacios de soluciones (véanse los Ejercicios 14.19 y 15.2).

Apartado (c)--------------------------------

Cuando hay varias riquezas de cada clase, la situación es equivalente a tener varios objetos con las mismas características olvidándonos de la clase a la cual pertenece cada objeto. Por tanto, como las riquezas de cada clase son divisibles, podemos utilizar la misma estrategia del Apartado (a) para elegir objetos, puesto que a la estrategia no le

afecta que haya objetos repetidos. Equivalentemente, al ser los objetos fraccionables, se puede considerar cada clase como un único “gran objeto" multiplicando los valores y pesos correspondientes por el número de objetos en la clase.

En resumen, ordenamos las clases de riqueza en orden decreciente de valor por unidad de peso, y se llena la mochila con tanta cantidad de la clase de turno como haya disponibilidad, en tanto lo permita la capacidad de la mochila. El algoritmo que implementa esta estrategia es el siguiente:

```
fun mochila-clases-v(P[1..n], V[1..n] de
real+, C[1..n] de nat+, M: real+) dev
sol[1..n] de real
var D{1..n] de real+, I[1..n] de 1..n {para
ordenar los índices}

  para i = 1 hasta n hacer
    D[i]:= P[i]/V[i]
  fpara

I:= ordenarIndices(D)
peso:= 0 ;
i:= 1
  mientras i≤n ∧c peso + P[I[i]] * C[I[i]≤M
  hacer
  {podemos coger la cantidad completa de riquezas
  de clase I[i]}
    sol[I[i]]:= C[I[i]];
    peso:= peso + P[I[i] * C[I[i]]
    i:= i + 1
  fmientras
```

```
si i≤n entonces {coger cierta cantidad de la
clase I[i]}
  sol[I[i]]:= (M - peso)/P[I[i]]
  para j= i+1 hasta n hacer
    sol[I[j]]:= 0
  fpara
fsi
ffun
```

De nuevo, el coste total del algoritmo es del mismo orden que la ordenación de las clases.

Apartado (d)-----------------------

Si la cantidad disponible de cada clase es inagotable, entonces es obvio que podemos llenar por completo la mochila con riquezas de una única clase: la de *mayor valor por unidad de peso*.

12.6. Monedas

Se dispone de un conjunto finito $M = \{m_0,\ldots,m_n\}$ de tipos de monedas, donde cada m_i es un número natural que indica el valor de las monedas de tipo i, y se cumple $m_0 < \ldots < m_n$. Suponiendo que la cantidad disponible de monedas de cada tipo es ilimitada, se quiere pagar de forma exacta una cantidad $C > 0$ utilizando un número mínimo de monedas. (En general, el problema puede no tener solución, porque C puede ser inalcanzable de forma exacta, pero si $m_0=1$ la existencia de solución está claramente garantizada).

a) Una estrategia voraz para encontrar una solución consiste en considerar los tipos de monedas de mayor a menor valor, e ir cogiendo tantas monedas de cada tipo considerado como sea posible. Escribir un algoritmo que implemente esta estrategia. Demostrar que, en general, esta estrategia no siempre da lugar a una solución óptima que minimice el número total de monedas.

b) Demostrar que si $M = \{v^0,\ldots,v^n\}$ para algún entero $v > 1$, la estrategia voraz propuesta en el apartado anterior da lugar a una solución óptima.

c) Demostrar que la estrategia voraz también es correcta en el caso más general en que cada tipo de moneda tiene un valor múltiplo del anterior, es decir, $m_0=1$ y para todo i entre 1 y n

se tiene que $m_i = v_{i-1}\, m_{i-1}$ para algún $v_{i-1}>1$.

d) Hasta ahora se suponía ilimitada la cantidad disponible de monedas de cada tipo. Demostrar que, cuando cada tipo de moneda tiene un valor múltiplo del anterior, la estrategia voraz sigue siendo correcta aunque se limite la cantidad de monedas disponibles de cada tipo.

-------------------Solución----------------

Apartado (a)

El siguiente algoritmo implementa la estrategia propuesta:

```
fun monedas-v(M[0..n] de nat+, C: nat) dev sol[0..n] de nat

sol[0..n]:= [0]
falta := C;
i:= n
mientras falta ≠0 ∧ i ≥0 hacer
   sol[i] := falta div M[i]
   falta := falta mod M[i]
   i := i - 1
fmientras
ffun
```

La condición $i \geq 0$, para no comprobar más tipos de monedas de los que tenemos, no es necesaria cuando $m_0=1$ pues si i llega a valer -1 tendremos *falta* = 0 ya que la última asignación habrá sido *falta* := *falta* mod 1.

Sin embargo, el algoritmo no siempre encuentra una solución óptima. Consideremos un conjunto de monedas de valores M = (1,4, 6) *y* una cantidad a

pagar C=8. El algoritmo anterior devolverá como solución pagar con una moneda de valor 6 y 2 monedas de valor 1, lo que hace un total de 3 monedas: mientras que la solución óptima es pagar con 2 monedas de valor 4.

En el Capítulo 13, dedicado a la técnica de programación dinámica, se vera cómo resolver este problema en el caso general (véase el Ejercicio 13.1).

Apartado (b)--------------------

Supongamos que tenemos un conjunto de tipos de monedas de la forma $M = \{v^0,\ldots,v^n\}$ para un cierto $v>1$. La demostración de la corrección de la estrategia voraz se apoya en el siguiente resultado:

Lema:

Para cualquier solución óptima $Y = (y_0,\ldots,y_n)$ se cumple que

$\forall i: 0 \le i < n : y_i < v.$ (12.1)

Demostración:

Si existiera algún i tal que $y_i \ge v$, entonces se cumpliría que $y_i = y'_i + v$ (con $y'_i \ge 0$) y por tanto $y_i v^i = y'v' + v^{i+1}$ con lo que se podría pagar la cantidad $y_i v^i$ con y'_i monedas de valor v^i y una moneda de valor v^{i+1}, y ya que $y'_i + 1 < y_i$, mejoraríamos la solución, lo cual contradice la optimalidad de Y.

Sea entonces $X = (x_0,\ldots,x_n)$, donde x_i es el número de monedas de valor v^i, la solución del algoritmo voraz. Sea $Y = (y_0,\ldots,y_n)$ una solución óptima, que existe

porque $v0$ =1 y el número de posibles soluciones es finito. Como X e Y son soluciones, se cumple que $\sum_{i=0}^{n} x_i v^i = C = \sum_{i=0}^{n} y_i v^i$. Comparamos X con Y empezando por las monedas de mayor valor, siendo j el primer tipo de moneda en el que difieran; entonces se tiene:

$$x_j v^j + \sum_{i=0}^{j-1} x_i v^i = C' = y_j v^j + \sum_{i=0}^{j-1} y_i v^i$$

No puede ocurrir que xj < yj porque el algoritmo voraz coge en cada paso el máximo posible de monedas del tipo correspondiente. Tampoco puede ocurrir que $x_j > y_j$ porque entonces no se puede compensar en el resto de Y la cantidad no pagada con monedas de valor v^j como demostramos a continuación. Formalmente, se tiene que

$$x_j > y_j \Rightarrow \sum_{i=0}^{j-1} y_i v^i \geq \sum_{i=0}^{j-1} y_i v^i - \sum_{i=0}^{j-1} x_i v^i$$
$$= (x_j - y_j) v^j > v^j$$

Por otra parte, por el resultado (12.1), se tiene que cada yi ≤ v - 1, entonces

$$\sum_{i=0}^{j-1} y_i v^i \leq \sum_{i=0}^{j-1} (v-1) v^i = (v-1) \sum_{i=0}^{j-1} v^i$$
$$= (v-1) \frac{v^{j-1} v - 1}{v-1} = v^j - 1 < v^j$$

con lo que se llega a una contradicción.

Concluimos entonces que $\forall j : 0 \leq j \leq n : x_j = y_j$, por lo que X e Y resultan ser la misma solución. Así pues, la solución generada por el algoritmo propuesto es

óptima.

Apartado (c)----------------

La demostración sigue exactamente el mismo esquema que en el apartado anterior, donde los valores eran potencias de un valor base. En este caso se verifica una propiedad análoga al resultado (12.1) del apartado anterior.

Lema:

Para cualquier solución óptima $Y = (y_0,...,y_n)$ se cumple que

$\forall i: 0 \leq i < n : y_i < v_i.$ (12.2)

Demostración:

Si $y_i \geq v_i$, entonces podemos escribir $y_i = v_i + y'_i$ con $y'_i \geq 0$, de donde $y_i m_i = v_i m_i + y'_i m_i = m_{i+1} + y'_i m_i$, por lo que se podrían sustituir $v_i > 1$ monedas de valor m_i por una moneda de valor m_{i+1}, contradiciendo la optimalidad de *Y.*

A continuación, verificamos el siguiente resultado:

Lema:

$\forall k: 1 \leq k < n: \sum_{i=0}^{k} (v_i - 1)m_i < m_{k+1}$ (12.3)

Demostración:

Por inducción sobre *k.* Como caso básico, para $k = 0$ tenemos $(v_0 - 1)m_0 < v_0 m_0 = m_1$.

Supongamos que $\sum_{i=0}^{k-1} (v_i - 1)m_i < m_k$; entonces

$$\sum_{i=0}^{k} (v_i - 1)m_i$$
$$= \sum_{i=0}^{k-1} (v_i - 1)m_i + (v_k - 1)m_k <^{h.i} m_k$$
$$+ (v_k - 1)m_k = v_k m_k = m_{k+1}.$$

Sea $X = (x_0,\ldots,x_n)$ la solución del algoritmo voraz para cambiar una cantidad C, que comparamos con una solución óptima $Y = (y_0,\ldots,y_n)$ (dicha solución existe). Sea $j \leq n$ el tipo de moneda de mayor valor tal que $x_j \neq y_j$. Como el algoritmo sigue la estrategia de coger el máximo número posible de monedas de cada tipo, tenemos $yj < xj$, lo que significa que en el resto de la solución óptima Y hay que compensar una cantidad $(xj - yj)mj \geq mj$ utilizando monedas de valores $m_0,\ldots,m_{j-1}$. Pero ya sabemos por (12.2) que $yi < vi$ (equivalentemente yi ≤ vi - 1) para *i* entre 0 y *j*-1 por lo que, utilizando el resultado (12.3),

$$\sum_{i=0}^{j-1} y_i m_i \leq \sum_{i=0}^{j-1} (v_i - 1)m_i < m_j$$

de modo que la compensación es imposible, y por tanto *X* debe ser igual a Y.

Apartado (d)-----------------------------

En primer lugar, hay que hacer notar que cuando limitamos el número de monedas disponible de cada tipo (*li* ≥ 0 para todo *i* entre 0 y n), puede que no

exista solución, incluso cuando haya monedas de valor 1. Para demostrar que la estrategia voraz propuesta sigue siendo óptima, demostraremos el siguiente resultado:

Si existe solución y hay una moneda que pudiendo ser utilizada no se utiliza, entonces es posible modificar la solución para utilizar dicha moneda, de forma que además se reduzca el número total de monedas usadas.

Esto es una consecuencia de la siguiente afirmación:

Lema:

Sea *M* = {m_0,…,m_n} donde para todo *i* entre 1 y *n* se tiene que mi = vi-1 mi-1 para algún entero vi-1 > 1. Sean xi ≥ 0, para todo *i* entre 0 y n, y sea *k* con *0 ≤ k ≤ n.* Entonces

V/ : *I* > 0 : ^~^x,ni, > *lnik+t* => *(Vi* : 0 < *i* < *k* : (3y, : 0 < y, < x; : *yiini* = /nn+i)). ,=0 <=ú

Demostración:

Por inducción sobre *k.* Como caso básico tenemos *k* = 0. Si xpnio > *lint* = /ionio, se sigue que -vo > *Ivo-* Basta entonces tomar yo = /t'o para obtener el resultado yonip = /romo = *lint.* Supongamos que el resultado es cierto para *k* — 1. y sea / > 0 tal que xjm* + x,ni, > Znn+i. Distinguimos 2 posibilidades:

1- -Vt>/Vi-

Nos basta tomar y;- = /t>i < x¿ e *yi* = 0 < x,. para tocio *i* entre 0 y *k*

— 1. obteniendo

k-1

yk">k + = *iVk"lk* + o =

/t»A-+l

í=0

*2' ** * ,Vk'*

A partir de la hipótesis x*mt + 52,=o — *l"'k* + \ *se t'c,le*

t-l

^2x,»t, *>/nit+t -Xfcitn =/vk"'k*

-Xk'iik = (Jvit -x_k)tn_k,

i=0

con *Ivk* — *Xk* > 0. Por hipótesis de inducción para *k* — I, existen 0 < y' < x,. para todo *i* entre 0 y *k* — 1. tales que

Á-I

Y^yí¹"' = (,ᵛk ~ Xk)"'k-

i=0

Tomamos yj = *x_k* e y, = *yi,* para todo *i* entre 0 y *k* — 1, con los cuales obtenemos

t-i Á-1

yk"ik + *^yi"ii =xknik* + ^y'm, =

xknik + *(ivk - x_k)»tk* = *Ivk>"k* =

l'"k+t-

1=0 í=0

Supongamos ahora que existe una solución *X* = (x_0,...,x_n) tal que

$$\exists k: 0 \leq k \leq n-1: l_{k+1} > x_{k+1} \wedge \sum_{i=0}^{k} x_i m_i > m_{k+1}$$

es decir, existe alguna moneda disponible del tipo m_{k+1} que, pudiendo haber sido utilizada en la solución, no

se ha utilizado. Tomando *l* = 1 el último lema nos permite afirmar que es posible quitar $\sum_{i=0}^{k} y_i > 1$ monedas de los tipos más pequeños y coger en su lugar una moneda del tipo m_{k+1}. Por tanto, si la solución no sigue la estrategia, entonces no es óptima, pues se puede ir mejorando a base de ir aplicando este resultado.

12.7. Ciudades invadidas

El enemigo, armado hasta los dientes con palos y piedras, ha desembarcado en barcas hinchables en las costas de nuestra patria invadiendo *n* ciudades. Los servicios de "inteligencia" están informados de que en cada una de las ciudades invadidas se encuentran *ei* efectivos enemigos. Para contraatacar, el Grupo de Intervención Rápida de Defensa dispone de *n* equipos listos para intervenir. Cada uno de estos equipos consta de *di* efectivos completamente equipados y entrenados. Para *garantizar* el éxito de la intervención en una ciudad es necesario que contemos al menos con tantos efectivos de defensa como el enemigo.

Describir e implementar una estrategia voraz que indique qué equipo de intervención debe ir a cada ciudad, de forma que se maximice el número de éxitos *garantizados.*

------------------------Solución--------------------------

Supondremos que las ciudades están numeradas de 1 a *n,* y que la información sobre el número de efectivos enemigos en cada ciudad la tenemos en un vector E[1..n] de forma que E[i] es el número de efectivos enemigos en la ciudad *i.* Igualmente la información sobre los equipos de defensa la tendremos en un vector D[1..n], de forma que D[i] es el número de efectivos de defensa en el equipo i.

La estrategia voraz que se propone consiste en ir repartiendo equipos de defensa de la siguiente manera: cuando tenemos que decidir qué equipo de defensa irá a la ciudad *i,* si hay todavía equipos no asignados con al menos tantos efectivos como E[i], elegimos de entre ellos el equipo con menos efectivos, *y* si ningún equipo de defensa no asignado tiene suficientes efectivos, entonces elegiremos el equipo con menor número de efectivos de entre todos los restantes.

Demostramos la optimalidad de la estrategia propuesta, por el método de reducción de diferencias: para ello comparamos la solución del algoritmo voraz $X = (x_1,\ldots,x_n)$. donde xi es el equipo de defensa que va a la ciudad *i,* con una solución óptima $Y = (y_1,\ldots,y_n)$ que sabemos que existe porque el número de soluciones es finito. Sea j la primera posición donde difieran ambas soluciones. En la solución Y el equipo x_j aparecerá en alguna posición $l > j$. Veamos los distintos casos que se pueden presentar:

1. Ambas soluciones garantizan la victoria en la ciudad j, es decir, $e_j \leq d_{xj}$. $\wedge$ $e_j \leq d_{yj}$. Como el algoritmo voraz asigna la menor defensa disponible, sabemos que $d_{xj} \leq d_{yj}$. Entonces, al intercambiar y_j con y_l en Y, la solución sigue siendo óptima, ya que en la ciudad j se sigue garantizando la victoria (se garantiza

en *X)* y en la ciudad *l* no ha podido empeorar la situación, ya que ahora mandamos un equipo con al menos tantos efectivos como antes.

2. La solución voraz garantiza la victoria pero la solución óptima no, es decir, $ej \leq d_{xj} \wedge ej > d_{yj}$. En tal caso, se tiene que $d_{yj} \leq d_{xj}$ y podemos intercambiar en la solución óptima *yj* con y_l. Así. logramos que se garantice la victoria en la ciudad *j* y, aunque habremos empeorado la situación en la ciudad l, la solución obtenida sigue siendo óptima ya que se mantiene el número de ciudades vencidas.
3. La solución del algoritmo no garantiza la victoria aunque la óptima sí, es decir, $e_j > d_{Xj} \wedge ej \leq d_{yj}$. Este caso no se puede presentar, porque el algoritmo voraz nunca hubiera asignado una defensa perdedora a la ciudad *j* si d_{yj}, que garantiza la victoria, estaba disponible.
4. Ninguna solución garantiza la victoria, es decir, $e_j > d_{Xj} \wedge e_j > d_{yj}$. Cuando no se puede garantizar la victoria, el algoritmo voraz asigna la menor defensa de entre las restantes, por lo que se tiene $d_{xj} \leq d_{yj}$. En este caso también podemos intercambiar en la solución óptima y_j con y_l, y la solución obtenida no puede ser peor que la de partida.

El algoritmo que implementa la estrategia es el siguiente:

```
fun asignar-equipos(E[1..n), D[1..n] de
nat+) dev cual-va[1..n] de 1..n
{cuál-va[i] será el equipo que vaya a la ciudad i}
var asignado[ 1..n ] de bool {indica los
equipos ya asignados}

asignado [1..n] := [falso]
para i = 1 hasta n hacer {recorrer las
ciudades}
   ⟨éxito, menor⟩:= buscarMenorGanador(D,
   asignado, E[i])
   si ¬éxito entonces {no hay ningún equipo que
   garantice la victoria}
      menor:= buscarMenor(D, asignado)
   fsi
   cual-va[i]:= menor;
   asignado[menor]:= cierto
fpara
ffun
```

La función buscarMenorGanador busca en el equipo no asignado con menor numero de efectivos que cumpla que tiene al menos tantos efectivos como E[i].

```
fun buscarMenorGanador(D[1..n] de nat⁺,
asignado[1..n] de bool, e : nat⁺) dev ⟨ b : bool, p
: nat⟩

i := 1
mientras i ≤ n Ac (asignado[i] v D[i] < e)
hacer
   i:= i + 1
fmientras

si i > n entonces
   b:= falso
si no
   b:= cierto;
   p:= i
   para j = i+1 hasta n hacer
      si ¬asignado[j] ∧ D[j]≥ e ∧ D[j]<D[p]
```

```
entonces
  p:= j
fsi
fpara
fsi
ffun
```

La función buscarMenor, que no detallamos, busca en *D* el equipo no asignado con menor número de efectivos.

Como el coste de buscarMenorGanador y buscarMenor es lineal con respecto al número de equipos, el coste de asignarEquipos es cuadrático con respecto al número de equipos que coincide con el de ciudades.

Una estrategia alternativa consiste en considerar los equipos defensores previamente ordenados de forma creciente.

Cuando tenemos que decidir a qué ciudad irá el equipo de defensa *i.* si hay todavía ciudades indefensas donde el enemigo no supere a D[í], elegimos cualquiera de ellas; pero si el equipo de defensa no tiene suficientes efectivos para defender las ciudades restantes, entonces elegiremos la ciudad con el mayor número de enemigos de entre todas las restantes.

Suponemos entonces que *d* < ... < *d„* y demostramos la optimalidad de esta nueva estrategia por el método de reducción de diferencias, comparando la solución del algoritmo voraz *X* = (AI . A?

A„).

donde *x,* es la ciudad a la que va el equipo de defensa *i,* con una solución óptima *Y* = (vi. ys, ... y_n). Sea *j* la primera posición donde difieran ambas soluciones. En la solución *Y* la ciudad A, aparecerá en alguna posición / > j y. por hipótesis, se tiene que *di* > *dj.* Veamos los distintos casos que se pueden presentar:

1. Ambas soluciones garantizan la victoria con el equipo *j.* es decir, *dj* > <■, (A *dj* > e, . En ese caso, podemos intercambiar las ciudades yy e *yi* para obtener una solución que sigue siendo óptima, ya que el equipo *j* sigue teniendo la victoria garantizada en la ciudad y/ = v, (está garantizada en *X)* y ahora, al enviare! equipo *I.* con *d/* > *dj* > e_y , se garantiza también la victoria en la ciudad yy.
2. La solución voraz garantiza la victoria, pero la solución óptima no. es decir. <7, > <-A A *dj* < e_{yi}. En este caso, al intercambiar las ciudades yy e y, logramos asegurar una nueva victoria en la ciudad *yi* — *Xj.* Aunque habremos empeorado la situación para el equipo /. la solución obtenida sigue siendo óptima ya que se mantiene el número de ciudades vencidas.
3. La solución del algoritmo no garantiza la victoria aunque la óptima

sí. es decir, $dj < e_{X/}$ A $dj > e_{yi}$. Este caso no se puede presentar, porque si la ciudad yy estaba indefensa y el equipo *J* garantiza allí la victoria, entonces el algoritmo voraz hubiera elegido dicha ciudad para el equipo *j.* antes de enviar a este a una posible derrota.

4. Ninguna solución garantiza la victoria, es decir, $di < e_x$ A $di < e_y$. En este caso también podemos intercambiar en la solución óptima yy con y/, ya que el algoritmo voraz asegura que la ciudad asignada al equipo *j* es la que tiene más enemigos de entre las restantes; por tanto, se tiene $e_{X)} > e_{yj}$ y la situación del equipo / en la solución modificada solo puede ser mejor que antes.

Para conseguir una implementación eficiente, nos interesa tener ordenados también de forma creciente los efectivos enemigos (ej < *e2 < ... < e„).* Ahora bien, como los efectivos enemigos ya están fijados en cada ciudad, al ordenar dichos efectivos, tendríamos que considerar la correspondiente reordenación de ciudades. Para simplificar la presentación del algoritmo, supondremos entonces que las ciudades ya vienen numeradas con el orden correspondiente a efectivos crecientes.

{D[1]≤ ... ≤ D[n] A E[1] < ... < E[n]}

fun asignar-equipos2(E[1..n], D[1..n] **de** *nat+)* **dev** *adónde-va[*1..n] **de** 1..n
{adónde-va[i] será la ciudad adonde va el equipo *i}*

```
j:= 1;
k:= n {las ciudades por defender son las numeradas desde j hasta k}
para i=1 hasta n hacer {recorrer las defensas}
   si E[j]≤D[i] entonces {se garantiza la victoria}
      adónde-va[i]:= j;
      j:= j + 1
   si no {no se garantiza la victoria y el equipo va a la última ciudad sin defensa}
      adónde-va[i]:= k;
      k:= k − 1
   fsi
fpara
ffun
```

Puesto que ahora el bucle es de complejidad lineal, la complejidad total del algoritmo está en el orden de la complejidad de la ordenación de los vectores.

12.8. Cine

La filmoteca ha organizado un maratón de cine de terror. Durante 24 horas se proyectarán películas (todas diferentes) en las *n* salas disponibles. Deborah, gran aficionada a este género de películas, ha conseguido la programación completa donde aparecen todas las películas que se van a proyectar durante el maratón; junto con el título, nombre del director, duración de la película y otros datos de interés, se indica la sala de proyección y la hora de comienzo. Ayudar a Deborah a planificar su maratón de cine, teniendo en cuenta que su único objetivo es ver el máximo número posible de películas.

--------------------------Solución-----------------------------

Tenemos un conjunto de *n* películas, cada una con un instante de *comienzo* c_k y una duración d_k, lo que permite calcular el correspondiente instante de *finalización*: $f_k = c_k + d_k$.

Por tanto, las películas *i y j* son ***compatibles*** si los intervalos [ci, fi) y [c_j, fj) no se solapan, es decir si $ci \geq fj$ *o* $cj \geq fi$.

Tenemos que seleccionar el mayor número de películas que sean mutuamente compatibles.

La estrategia VORAZ consiste en considerar las películas por *orden creciente* de instante *de finalización*, y en cada paso *i* seleccionar la película í-ésima si no solapa con ninguna de las ya elegidas, y rechazarla en caso contrario.

Para no perder la relación entre los

índices en los vectores de entrada *C* y *D* y los tiempos de finalización correspondientes en el vector F, este no se ordena directamente sino utilizando la función ordenar-indices del Ejercicio 11.6.

```
fun peliculas(C[1..n] de nat, D[1..n) de nat⁺)
dev sol[1..n] de bool
var F[1..n] de nat⁺      {tiempos finales}
    I[1..n] de 1..n      {para ordenar los índices}

  para i = 1 hasta n hacer
    F[i] := C[i] + D[i];
    sol[i]:= falso;
  fpara

I:= ordenarIndices(F)   {F[I([1] ≤ … ≤ F[I[n]]}
sol[I[1]]:= cierto {la primera película se elige
siempre}
final:= F[I[1]]  {final de la última película elegida}

  para i = 2 hasta n hacer
    si C[I[i]] ≥ final entonces
      sol[I[i]]:= cierto;
      final:= F[I[i]];
    fsi
  fpara
ffun
```

Como las películas se eligen en orden creciente de tiempo de finalización, *final* es siempre el máximo de los tiempos de finalización de las películas ya elegidas, y. por tanto, para comprobar si la película *i* es compatible con las ya elegidas es suficiente comprobar si *c,* no es menor que *final.* De esta forma, el bucle voraz es de complejidad lineal respecto al número de películas, y el orden de complejidad del algoritmo completo corresponde al de la ordenación del vector de tiempos finales.

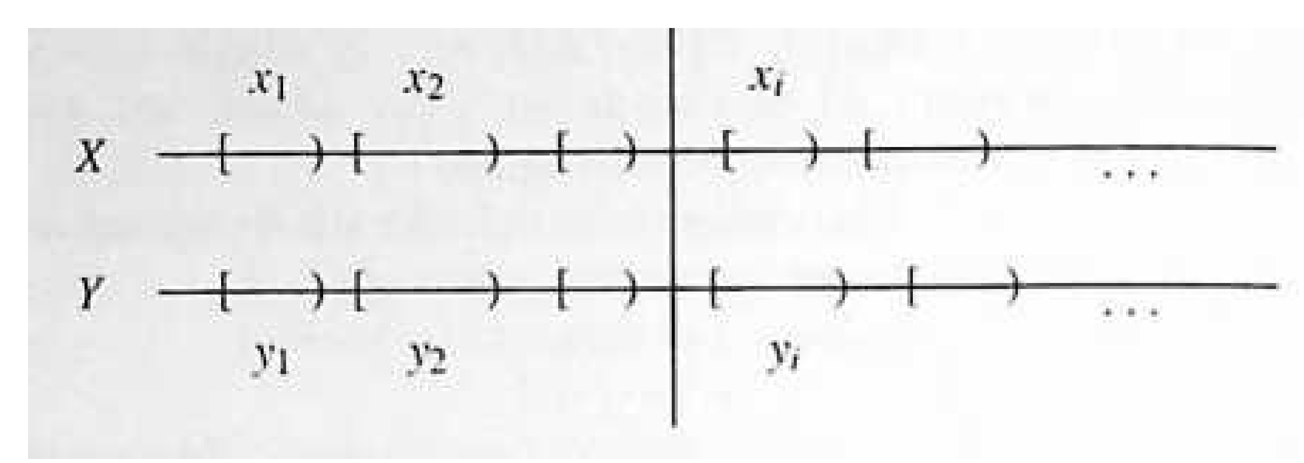


Figura 12.1: Reducción de diferencias para el problema del maratón de cine.

Demostramos ahora que la estrategia es óptima, mediante el método de *reducción de diferencias*, comparando la solución obtenida por el algoritmo, *X,* con una solución óptima, *Y,* que existe por ser finito el número de soluciones. Suponemos que ambas están ordenadas por tiempo de finalización creciente. Sea *i* la primera posición donde ambas soluciones difieren, como muestra la Figura 12.1. Puesto que el algoritmo voraz ha elegido *xi* y no yi, sabemos que $f_{Xi} \leq f_{Yi}$. Podemos entonces sustituir la película *yi* por la película xi en la solución *Y* para obtener otra solución también óptima. Es solución porque la película xi no solapa con ninguna de las demás películas en *Y*: no solapa con las anteriores porque no lo hace colocada en *X* (y las películas anteriores a la *i* son iguales en ambas soluciones) y no lo hace con las siguientes porque no lo hacía yi y *xi* termina antes. La solución *Y* sigue siendo óptima porque el número de películas no varía al hacer el intercambio.

Siguiendo con este proceso, podemos hacer que todas las películas seleccionadas por el algoritmo voraz

aparezcan en una solución óptima. Además, la solución óptima así obtenida no tiene más películas que X porque, en otro caso, al tener dichas películas un tiempo de finalización mayor, el algoritmo voraz las habría considerado y seleccionado.

12.9. Conferencias

La Universidad Imponente tiene que planificar un evento cultural que consiste en *n* conferencias. Para cada conferencia se conoce la hora de comienzo y la de finalización fijadas por los ponentes. Se desea planificar las *n* conferencias distribuyéndolas entre las distintas salas disponibles, de forma que no haya 2 conferencias en una misma sala al mismo tiempo. El objetivo es minimizar el número de salas utilizadas, para así causar el menor trastorno al resto de las actividades académicas.

---------------------------Solución---------------------------

Suponemos que cada conferencia viene definida por un intervalo de la forma [ci, fi). La estrategia voraz que proponemos consiste en considerar las conferencias por *orden creciente de tiempo de comienzo*. A continuación, se considera cada conferencia por turno; se intenta planificar en alguna sala previamente utilizada, y si ello no es posible, se le asigna una sala “nueva". Nótese que solo se exige que la conferencia de turno no solape con ninguna de las conferencias previamente asignadas a la sala considerada, por lo que si hubiera varias salas que cumplieran dicha condición, sería indiferente la sala finalmente elegida.

Demostraremos la optimalidad de la estrategia, por inducción sobre el número de conferencias. En el caso básico, con $n=1$, obviamente se necesita una única

sala. Supongamos como hipótesis de inducción que la estrategia es óptima para *n* - 1 conferencias para las que necesita *k* salas. Consideramos la conferencia n-ésima, definida por el intervalo $[c_n, f_n)$. Si esta conferencia se puede planificar en alguna de las *k* salas, la estrategia voraz produce una solución óptima. Si no es así (es decir, si la estrategia voraz necesita utilizar una sala más) ¿sigue siendo óptima la solución? ¿No hay manera de reubicar las conferencias de forma que *k* salas sean suficientes? Veamos que no. Si la conferencia *n* es incompatible con todas las salas, es porque en cada sala existe una conferencia que solapa con la conferencia *n*. Además, ninguna de estas conferencias empieza más tarde que la conferencia *n* (puesto que las vamos considerando en orden). Es decir, existen $[ci_1,fi_1),\ldots,[ci_k, fi_k)$ tales que $ci_j \leq c_n < fi_j$ para todo *j* entre 1 y *k*. Por tanto tenemos la situación de la Figura 12.2, es decir, *k*+1 conferencias que solapan en el tiempo, lo que implica que son necesarias *k* + 1 salas.

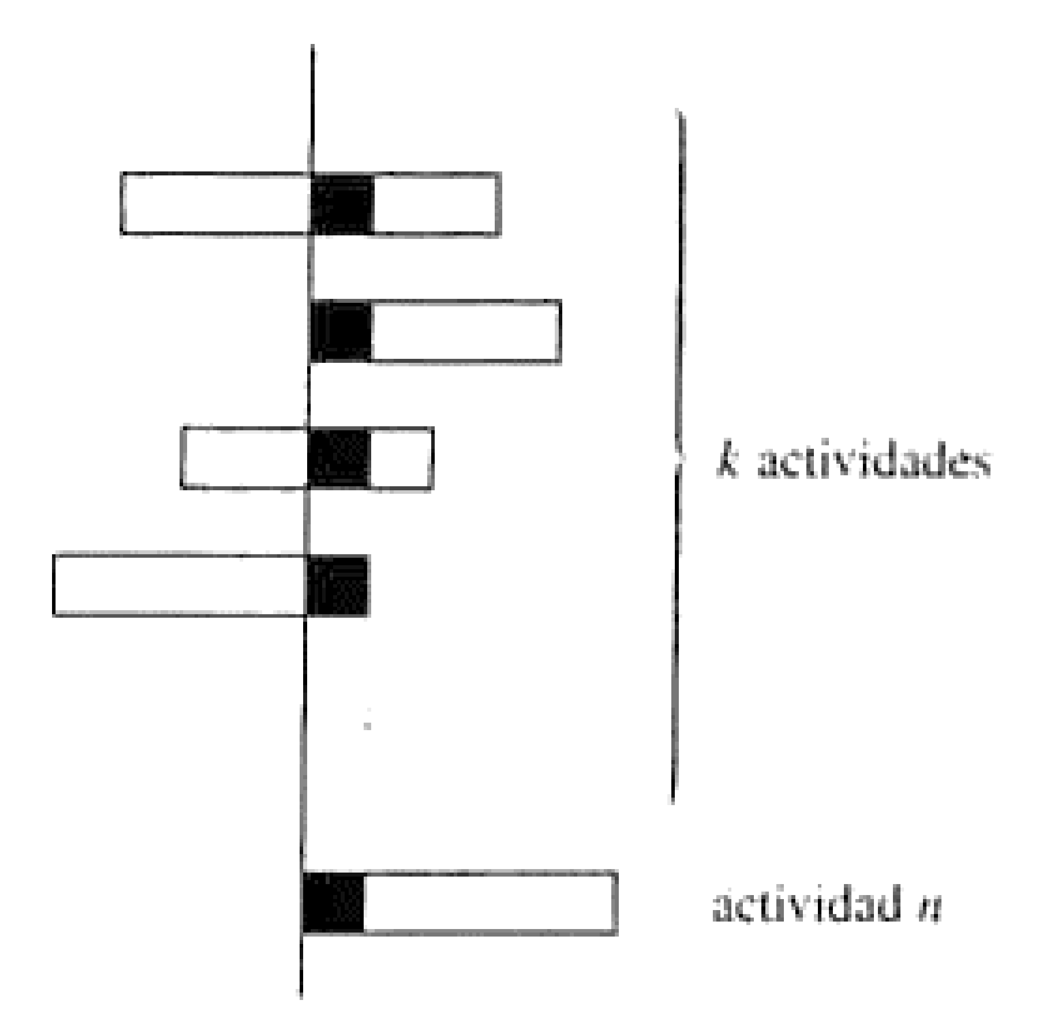


Figura 12.2: Minimizando el número de salas ocupadas.

Suponiendo que las conferencias vienen dadas por orden creciente de tiempo de comienzo, el algoritmo que implementa la estrategia es el siguiente:

{C[1]≤…≤C[n]}

```
fun conferencias(C[1..n] de nat, F[1..n]
de nat+) dev ⟨sala[1..n] de nat, k: nat)⟩
{sala[i] es la sala donde se planifica la conferencia i,
para i entre 1 y k}
{ k es el número de salas utilizadas }
var tiempo[1..n] de nat+ {indica hasta cuándo
está ocupada cada sala}
k := 0
  para i = 1 hasta n hacer
    {buscar una sala libre}
    j := 1
    mientras j ≤ k ∧c tiempo[j] > C[i]
    hacer
      j := j+1
    fmientras
```

```
      si j ≤ k entonces  {se planifica en una sala ya utilizada}
        sala[i] := j
        tiempo[j] := F[i]
      si no   {se ocupa una sala nueva}
        k := k + 1
        sala[i] := k
        tiempo[k] := F[i]
      fsi
   fpara
ffun
```

El coste de este algoritmo es cuadrático respecto al número de conferencias.

12.10. Viajante

Un viajante de comercio tiene que viajar en coche desde Valencia a Lisboa siguiendo una ruta preestablecida. Con el depósito lleno, su coche puede recorrer un máximo de ***M* kilómetros**. El viajante tiene un mapa de carreteras en el que figuran las *distancias* entre las gasolineras en su ruta, y desea utilizar esta información para, suponiendo que parte de Valencia con el depósito lleno, realizar en su recorrido un número mínimo de paradas para repostar combustible. Desarrollar un algoritmo eficiente para determinar en qué gasolineras deberá parar.

---------------------------Solución-----------------------

Suponemos las gasolineras de la ruta numeradas de 0 a *n,* siendo la gasolinera g_0 la correspondiente a Valencia y g_n la de Lisboa, y que las distancias entre gasolineras adyacentes vienen dadas en un vector D[1..n] de forma que D[i] = *distancia*(g_{i-1}, g_i) para todo *i* entre 1 y *n.* Suponemos que todas la distancias D[i] entre gasolineras adyacentes en la ruta son menores o iguales que *M,* pues en otro caso el problema no tiene solución.

La estrategia voraz consiste en no parar mientras no haga falta; es decir, al llegar a cada gasolinera, si hay gasolina suficiente en el depósito para llegar hasta la siguiente se continúa, y si no, se para y se llena el depósito.

Aplicando el método de reducción de diferencias, vamos a demostrar que la

solución obtenida por la estrategia voraz es óptima, es decir, que minimiza el número de paradas. Para explicar la transformación de una solución óptima en la solución generada por la estrategia, consideramos los esquemas de la Figura 12.3, donde los puntos representan las gasolineras a lo largo de la ruta y los que tienen un circulo alrededor representan las paradas. Supongamos que ambas soluciones son distintas, pero que hacen la misma paradas hasta la gasolinera *g* incluida. Denotamos por *h* e *i* las siguientes gasolineras donde paran la solución óptima y la de la estrategia, respectivamente.

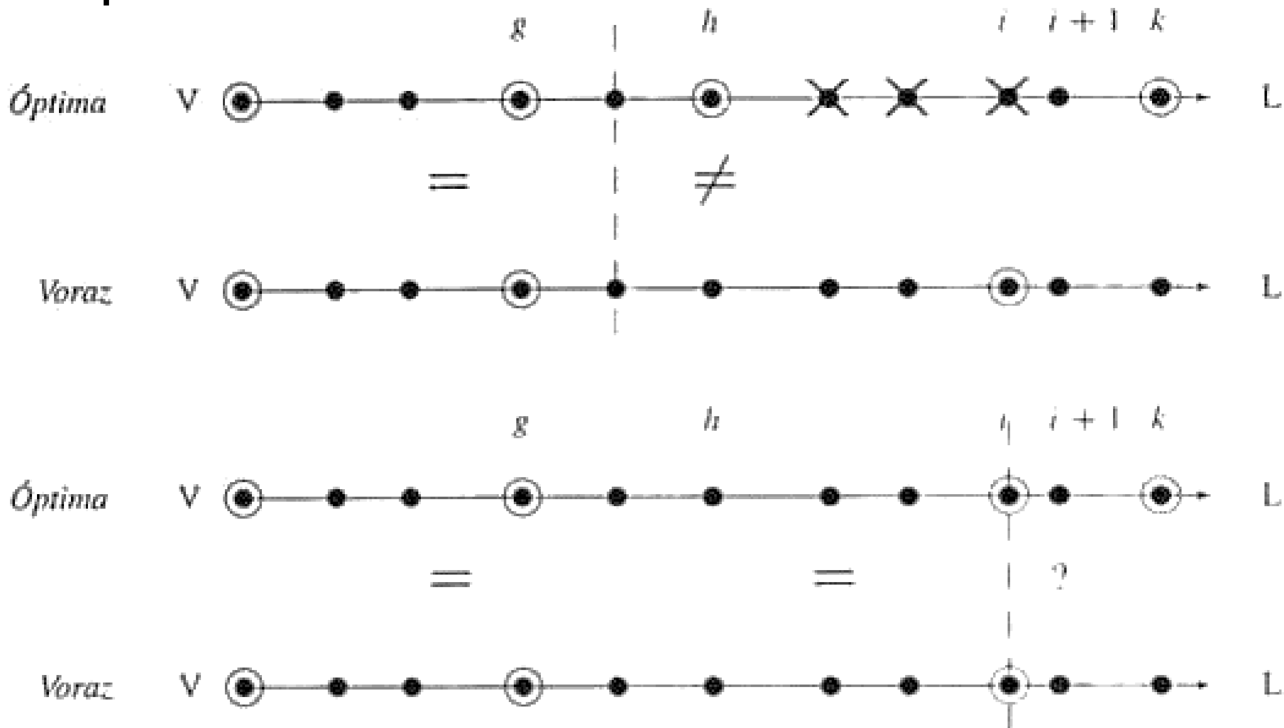


Figura 12.3: Reducción de diferencias para el problema de las gasolineras.

En primer lugar, observemos que la situación reflejada en el primer esquema es correcta, pues *h* estará situada antes que *i* en la ruta, dado que la estrategia voraz no para mientras ello no sea imprescindible.

En segundo lugar, es claro que la

solución óptima no puede parar en las gasolineras entre la *h* y la *i,* incluyendo esta última, lo cual se representa en la figura mediante una cruz (x) sobre dicha gasolineras. La razón es que si parase en alguna de ellas, realizaría al menos 2 paradas para recorre un trayecto que es posible recorrer con una sola parada, como demuestra la solución voraz, por lo que la parada en *h* se podría evitar, contradiciendo la optimalidad de la solución.

La transformación que llevamos a cabo en la solución óptima (segundo esquema) es no parar en *h* para hacerlo en *i.* Ello es posible, pues *distancia(g,*i) ≤ *M* y siendo *k* la parada siguiente a *h* en la solución óptima, antes de la transformación, se cumple que *distancia(i,k)* ≤< *distancia(h,k)* ≤ *M.* Con esta transformación hemos conseguido que ambas soluciones sean iguales al menos hasta la gasolinera *i.* Como en la solución óptima no hemos cambiado el número de paradas, se preserva la optimalidad.

El algoritmo que implementa la estrategia propuesta es el siguiente:

{ Vi*i* : 1 < *i* < *n* : D[í] < *M*)

```
fun gasolineras(D[1..n] de nat⁺, M : nat⁺)
dev (G[0..n] de bool, paradas : nat)

G[1..n] := [falso]
G[0] := cierto {sale con el depósito lleno}
paradas := 0;
km := D[1]
para i = 1 hasta n-1 hacer
```

```
    si km + D[i+1] ≤ M entonces
      G[i] := falso
    si no
      G[i] := cierto
      paradas := paradas + 1;
      km := 0
    fsi
    km := km + D[i+1]
  fpara
ffun
```

El coste de este algoritmo es lineal con respecto al número de gasolineras.

12.11. Huertas

El tío Facundo posee *n* huertas, cada una con un tipo diferente de árboles frutales. Las frutas ya han madurado y es hora de recolectarlas. La recolección de una huerta exige un día completo. El tío Facundo conoce, para cada una de las huertas, el beneficio que obtendría por la venta de lo recolectado. También sabe los días que tardan en pudrirse los frutos de cada huerta.

(a) En estos casos, el *manual del buen recolector* sugiere utilizar una estrategia voraz. Ayudar al tío Facundo a decidir qué debe recolectar y cuándo debe hacerlo, para maxitnizar el beneficio total obtenido.

(b) Estudiar si la estrategia utilizada para solucionar el apartado anterior es válida también en el caso de que la recolección de cada huerta requiera un número arbitrario de días.

-------------------Solución-------------------

Apartado (a)

Cada una de las *n* huertas lleva asociado un beneficio positivo *b,* que solo podrá contabilizarse si la huerta es recolectada sin superar el plazo correspondiente *pi.* Por ejemplo, si una huerta tiene plazo 3. solo puede ser recolectada en los días 1. 2 y 3.

Diremos que un conjunto de huertas es ***factible*** si existe alguna secuencia de recolección admisible que permita recolectar todas las huertas del conjunto

dentro de sus respectivos plazos. Es decir, una permutación $(i_1,\ldots,i_k)$ de las huertas de un conjunto es ***admisible*** si $\forall j$: $1 \leq j \leq k : p_{ij} \geq j$ (el plazo correspondiente es posterior al momento en que se recolecta).

La estrategia voraz que proponemos consiste en ir seleccionando las huertas de forma que, en cada etapa, se escoge la huerta aún no considerada con *mayor beneficio*, con la condición de que el subconjunto formado siga siendo *factible*.

Demostramos la optimalidad de esta estrategia utilizando el método de *reducción de diferencias*; para ello, supondremos que las huertas están ordenadas por beneficios, y que la estrategia voraz obtiene un subconjunto I de índices, mientras que la solución óptima considerada consiste en un subconjunto J de índices. Sean S_I y S_J secuencias admisibles de las huertas de los subconjuntos I y J. respectivamente (existen porque ambos subconjuntos son factibles). En general habrá huertas comunes en I y en J, entonces primero transformamos S_I y S_J de tal forma que dichas huertas comunes se recolecten en los mismos días, obteniendo secuencias S'_I y S'_J. Esto es siempre posible, ya que si la huerta α se recolecta más tarde en S_J que en S_I, podemos modificar S_I de la siguiente manera: si el día d en el que la huerta α se recolecta en S_J no se recolecta ninguna huerta en S_I, podemos

retrasar la recolección de α hasta el día *d* en la secuencia S_I, obteniendo otra secuencia también admisible; si en el dia *d* ya se recolecta una huerta β en *Si,* podemos intercambiar las huertas α y β obteniendo otra secuencia admisible, ya que la recolección de toda huerta se puede adelantar. En el caso de que una huerta α se recolecte antes en S_I que en Sj, podemos hacer la misma modificación sobre *Sj.* Obsérvese que en ningún caso se modifica la solución de la estrategia voraz, que es el subconjunto I, y no la secuencia S_I.

Comparemos ahora las secuencias S'_I y S'_J hasta encontrar una posición *k* donde difieran. No puede ocurrir que en Sj se recolecte una huerta en el día *k y* ninguna en *S'j* porque añadiendo dicha huerta a *J* obtendríamos un conjunto factible con una suma de beneficios mayor, *y J* es óptima. Tampoco puede ocurrir que se recolecte una huerta en S'_J *y* ninguna en S'_I, porque la estrategia voraz, al considerar dicha huerta la habría añadido a *I,* ya que había hueco para recolectarla. Es decir, que el día *k* se recolectan una huerta α en S'_I y otra β en S'_J. Veamos qué relación puede haber entre los beneficios b_α y b_β. Si fuera mayor el beneficio b_α, cambiando α por β en *J* obtendríamos una solución mejor y eso no es posible, ya que *J* es óptima. Tampoco puede ocurrir que $b_\beta > b_\alpha$ porque la estrategia considera las huertas en

orden decreciente de beneficios y habría considerado antes β que *α*. Es decir, que el único caso posible es que $b_\alpha = b_\beta$.

Deducimos por tanto que. si S'_I y S'_J recolectan huertas distintas en un mismo día, estas huertas tienen que tener el mismo beneficio y, por tanto, se puede cambiar β por α en *J* y el beneficio total de ambas soluciones es el mismo. Sucesivas transformaciones de este estilo nos llevan a una solución óptima igual que la solución voraz.

Para determinar si un conjunto de huertas es factible, basta con encontrar una secuencia de recolección admisible. Podríamos ir calculando permutaciones hasta encontrar una admisible, pero vamos a demostrar que es suficiente comprobar la secuencia que ordena las huertas correspondientes por plazos. Para ello, utilizamos el siguiente

Lema:

> Un conjunto *H* de huertas es factible si y solo si la secuencia de las huertas de *H* ordenadas de forma creciente según plazo es admisible.

Demostración:

La implicación hacia la izquierda es trivial, así que nos limitamos a demostrar la implicación hacia la derecha, lo que se hará por contrarrecíproco.

Sea $H=\{h_1,\ldots,h_k\}$ con $p_1 \leq \ldots \leq p_k$ tal que la secuencia $h_1,\ldots,h_k$ no es admisible, es decir, existe alguna huerta h_r con $r \in \{1,..,k\}$ tal que $p_r < r$. Pero entonces se cumple que $p_1 \leq \ \ldots \leq P_{r-1} \leq P_r \leq r-1$ lo

que muestra que hay *r* huertas cuyos plazos son menores o iguales que *r*–1, y por tanto es imposible recolectar todas las huertas dentro de su plazo, por lo que se concluye que el conjunto *H* no es factible.

Por tanto, el test de factibilidad consistirá en comprobar si la secuencia de huertas ordenadas por plazos es admisible. En la implementación del algoritmo, aunque suponemos que las huertas vienen ordenadas por beneficio decreciente, las huertas seleccionadas para su recolección se mantendrán ordenadas por plazo creciente. Todo esto da lugar al siguiente algoritmo:

{B[1] ≥ ... ≥ B[n]}

```
fun huertas-v1 (P[1..n], B[1..n] de nat⁺)
dev ⟨huerta[1..n] de 1..n, k: 1..n⟩
{huertai[i] es la huerta recolectada en el día i, para i
entre 1 y k}
{k es el número total de huertas recolectadas}
var aux[1..n] de 1..n {secuencia auxiliar,
ordenada por plazos}

k:= 1;
aux[1]:= 1 {la primera huerta siempre se
recolecta}
para i=2 hasta n hacer
    {buscar hueco, lo más tarde posible}
    d:= k
    mientras d>0 ∧c (P[aux[d]] > P[i] ∧
    P[aux[d]]>d) hacer
        d:= d-1;
    fmientras

    {d=0 v P[aux[d]] < máx(P[i],d)}
    si P[i]>d entonces {puede recolectarse}
        {desplazar un día las huertas posteriores}
        para j=k hasta d+1 paso - 1 hacer
            aux[j+1]:= aux[j];
        fpara
        aux[d+1]:=i;
        k:= k+1;
    fsi
fpara
huerta[1..k]:= aux[1..k];
ffun
```

Al margen de la ordenación de las huertas por beneficio decreciente (que puede hacerse con un coste en $\Theta(n \log n)$), el coste del algoritmo anterior está en $\Theta(n^2)$ ya que en el caso peor hay que recolectar todas las huertas, y para cada una de ellas desplazar (un lugar) todas las anteriores; y esto en realidad es equivalente a una ordenación *por inserción* de las huertas por plazo creciente.

A continuación presentamos una forma alternativa para determinar la factibilidad de un conjunto de huertas, que va a permitir una implementación más eficiente de la estrategia voraz propuesta, y que está basada en el siguiente

Lema:

Un conjunto *H* de huertas es factible si y solo si la secuencia de las huertas de *H* donde estas se recolectan "lo más tarde posible", es admisible.

Demostración:

Comenzamos formalizando el concepto de "lo más tarde posible", que se refiere a que para cada huerta se elige el día libre más tardío y que no se pase de plazo; es decir, que para cada huerta h_i, el día elegido será

$$d(i) = \text{máx}\{d \mid 1 \leq d \leq min\{n, p_i\} \wedge (\forall j{:}\, 1 \leq j < i{:}\, d \neq d(j))\}.$$

Como en el lema anterior, la implicación hacia la izquierda es trivial y demostraremos la implicación hacia la derecha por contrarrecíproco. Supongamos que existe alguna huerta en *H* tal que todos los días antes de que expire su plazo *p* están ocupados. Para *l* = mín{n,p} sea *r* > *l* el primer día libre; entonces en *H* hay al menos *r* huertas (*r*-1 puestas más la que se está intentando colocar) con plazo menor que *r*, siendo por tanto imposible recolectar todas las huertas de *H* dentro de su plazo, por lo que el conjunto *H* no es factible.

Para implementar este test de factibilidad definimos una relación de equivalencia sobre los días de

recolección. Si definimos *libre(i)* = máx{d≤*i* | *d* libre}, es decir, el primer predecesor libre de *i,* entonces *i* y *j* estarán en la misma clase si y solo si *libre(i)=libre(j).* Para que *libre(i)* esté definido incluso si todos los días están ocupados, se considera también el día 0, que permanecerá siempre libre. La idea es que cada huerta hi debería recolectarse en el día *libre(pi),* porque representa el último día libre que respeta su plazo.

Obviamente, *libre(i)* va cambiando a medida que se va planificando la recolección de las huertas. Inicialmente, como no se ha planificado ninguna recolección, se tiene ∀i: 1≤i≤*n : libre(i)* = *i* (cada día está en una clase de equivalencia diferente); si recolectamos la huerta *h,* en el día *libre(pi)* (siempre que este no sea 0), al ocupar dicho día hay que *fusionar* la clase de equivalencia a la que pertenece *libre*(p_i) con la correspondiente al día anterior.

Para obtener una implementación eficiente de esta relación de equivalencia dinámica recurrimos a una estructura de *partición,* especificada en el Ejercicio 9.16. Como la implementación de la estructura de partición (véase el Ejercicio 9.17) no garantiza que el representante de la clase que devuelve la operación buscar sea el mínimo, que es lo que necesitamos en este algoritmo, vamos a utilizar un vector L|0..n] tal que L[í] =

libre{i] para todo *i* que sea representante de una clase de equivalencia.

En el algoritmo siguiente nos limitamos a considerar los días que no sobrepasen los plazos máximos de forma que. aunque declaremos el conjunto base de la partición como {0,…,n}, la parte útil es {0,…,*l*} para l = mín{n, $\text{máx}_{1\leq i\leq n}\{p_i\}$}.

(B[l]>B[2]>…>B[nj)

```
fun huertasV2(P[1..n], B[1..n] de nat⁺) dev
⟨huerta[1..n] de 1..k, k: 1..n⟩
var L[0..n], aux[1..n] de 0..n, partición:
partición[0..n]

l ;= P[1];
para i=2 hasta n hacer
  l := máx(l, P[i]);
fpara

l := mín(n, l);
partición:= crearPartición3()
aux[1..l]:= [0];
para i=0 hasta l hacer
  L[i]:= i;
fpara

para i=1 hasta n hacer
  c1:= buscar3(P[i],partición);
  m:= L[c1];
  si m≠0 entonces
    aux[m]:= i;
    c2:= buscar3(m-1, partición);
    fusionar3(c1, c2, partición);
    L[c1]:= L[c2];
  fsi
fpara

{comprimir solución}
k:= 0;
```

```
para i=1 hasta l hacer
  si aux[i]>0 entonces
    k:= k+1;
    aux[k]:= aux[i];
  fsi
fpara
huerta[1..k]:= aux[1..k];
ffun
```

El coste de la fase de inicialización de este algoritmo, incluyendo la creación de la partición, está en $\Theta(n)$. En cuanto al coste del bucle principal, se observa que en el peor de los casos, cuando se planifican todas las huertas, se realizan 2n búsquedas en la partición y *n* fusiones (hasta obtener una única clase de equivalencia). Como ya se detalló en el Ejercicio 9.17, la representación aquí utilizada garantiza que el coste de esta secuencia de *3n* operaciones es prácticamente lineal en *n.* Por supuesto, no debemos olvidar la ordenación previa de las huertas, lo que en total nos deja un algoritmo de coste en $\Theta(n \log n)$.

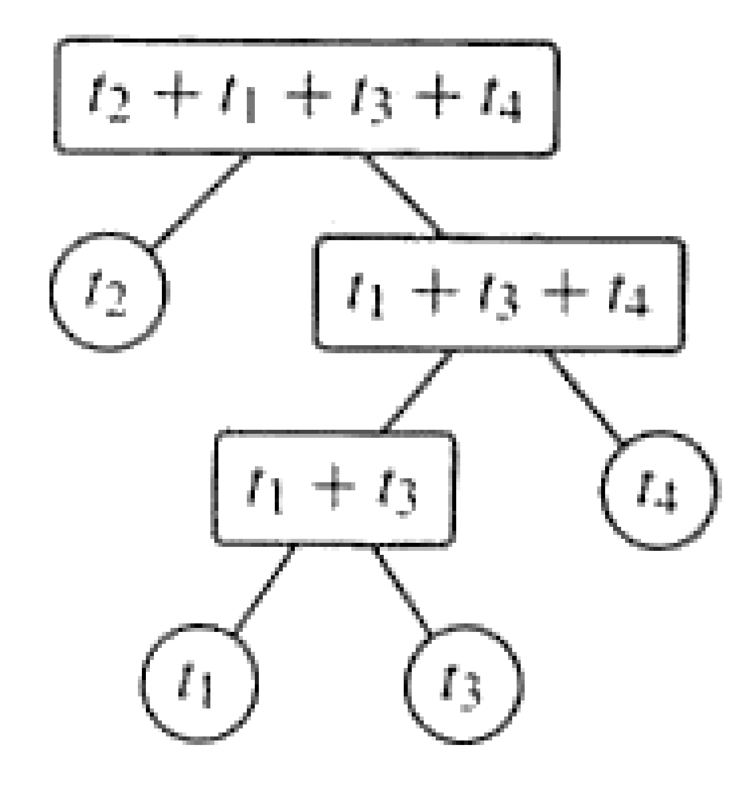


Figura 12.4: Ejemplo de árbol de mezclas.

Apartado (b)---------------------------

Vamos a suponer que el beneficio de una huerta solo puede contabilizarse si la huerta se termina de recolectar dentro del plazo correspondiente; es decir, si una huerta tiene plazo 4 y su recolección dura 2 días, entonces ha de empezar a recolectarse en los tres primeros días.

La estrategia propuesta en el apartado anterior ya no es válida en este caso, como ilustra el siguiente contraejemplo: tenemos 3 huertas, todas con plazo 4. pero mientras la primera tarda 3 días en recolectarse, las otras solo necesitan 2 días. Supongamos que los beneficios son 10, 8 y 6. respectivamente. Entonces, la estrategia anterior escogerá la primera huerta para empezar a recolectarla el primer día. por lo que las otras dos huertas quedarán sin recolectar y el

beneficio se reduce a 10. Sin embargo, es posible recolectar las otras dos huertas (en cualquier orden) y obtener un beneficio de 14.

En los Capítulos 14 y 15 dedicados a las técnicas de exploración de espacios de soluciones (vuelta atrás y ramificación y poda) se detallan soluciones alternativas para este problema (véanse los Ejercicios 14.16 y 15.4).

12.12. Mezclar lotes

El Maestro Piero, profesor de musicología, guarda las fichas de todos los alumnos que ha tenido a lo largo de los últimos *n* años. Para cada año, el lote de fichas está ordenado alfabéticamente. Pero ahora quiere reunir todos los lotes en uno solo, igualmente ordenado (se supone que todos los lotes son disjuntos, pues los alumnos se matriculan una sola vez con él). Para obtener el lote conjunto, el Maestro ha de ir mezclando (de forma ordenada) pares de lotes de fichas. Pero, como el tiempo empleado en la mezcla ordenada depende de los tamaños de los lotes a mezclar, no da lo mismo mezclar unos antes que otros. Encontrar una estrategia que determine el orden en el que se han de mezclar los lotes para minimizar el trabajo total de mezcla.

----------------------------Solución---------------------------

Cada posible orden de mezclas de los lotes puede representarse mediante un árbol binario con n hojas, correspondientes a los *n* lotes iniciales. Cada nodo interno corresponde al lote resultado de mezclar (ordenadamente) los dos lotes correspondientes a los hijos (izquierdo y derecho). Este tipo de árboles se denominan *árboles (binarios) de mezclas.* La Figura 12.4 muestra uno de estos árboles.

Como el coste de mezclar 2 lotes ordenados de fichas es proporcional a la suma de los tamaños de los lotes a mezclar, se tiene que el tamaño de un

lote determinado contribuye al coste total tantas veces como dicho lote se vea involucrado en una mezcla, o lo que es lo mismo, una menos del número del nivel en el que se encuentre la hoja correspondiente en el árbol de mezclas. Por tanto, si el tamaño del lote í-ésimo es *ti* (para *i* entre 1 y n), el coste total de una solución es

$$\sum_{i=1}^{n} t_i(\text{nivel}(i) - 1)$$

que denominamos ***suma ponderada de la frontera***. Así, nuestro problema se reduce a encontrar el árbol de mezclas que contiene en sus hojas a los elementos de un conjunto dado de números naturales y que minimiza dicha suma ponderada. Para resolver esta cuestión la estrategia que proponemos consiste en elegir, en cada momento, los dos lotes de menor tamaño para mezclarlos. El lote resultante de la mezcla pasa, a su vez. a formar parte de los lotes restantes a mezclar. La Figura 12.5 muestra el árbol de mezclas construido por la estrategia para lotes de tamaños 10, 15, 18. 22 y 35. Los números a la izquierda de los nodos internos indican el orden en el que se realizan las mezclas.

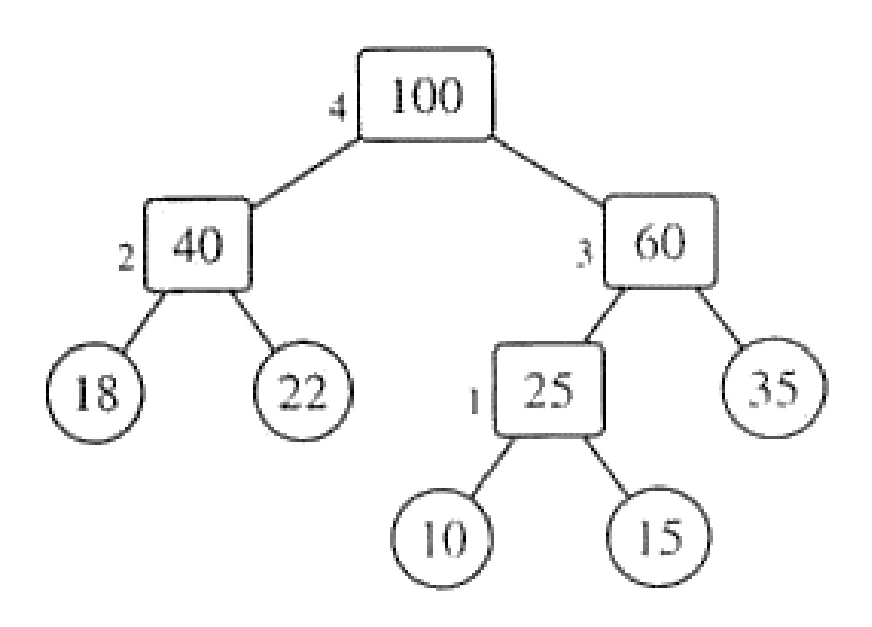


Figura 12.5: Ejemplo de árbol de mezclas construido por la estr

La demostración de que esta estrategia obtiene una solución óptima se hace por inducción sobre *n,* el número de lotes a mezclar. Cuando $n=1$, el lote único no necesita ser mezclado, por lo que no hacer nada es trivialmente óptimo. Supongamos que la estrategia obtiene árboles de mezclas óptimos para cualquier conjunto de n-1 lotes y vamos a demostrar que también obtiene una solución óptima para los lotes $t_1,\ldots,t_n$. Podemos asumir, sin pérdida de generalidad, que $t_1 \leq \ldots \leq t_n$; entonces t_1 y t_2 serán los lotes escogidos en la primera etapa de la estrategia y se creará un árbol de mezclas A con raíz t_1+t_2 y con los 2 primeros lotes como hijos. A continuación se obtendrá un árbol de mezclas *A'* para los lotes t_1+t_2, $t_3,\ldots,t_n$ y el resultado final *A"* se obtiene sustituyendo en *A'* la hoja que contiene t_1+t_2 por el árbol *A* anterior.

Si $L(A')$ es la suma ponderada de la frontera para el árbol A', se tiene que $L(A'') = L(A') + t_1 + t_2$ independientemente de dónde aparezca

en *A′* la hoja correspondiente a $t_1 + t_2$. Por hipótesis de inducción, el árbol *A′* es óptimo para $t_1+t_2, t_3,…,t_n$; así que si demostramos que la decisión de mezclar los lotes t_1 y t_2 siempre es correcta para la minimización de las mezclas, habremos demostrado que *A″* es óptimo para $t_1,…,t_n$.

Sea *B* un árbol de mezclas óptimo para los lotes $t_1,…,t_n$ y supongamos que en *B* no se mezclan los lotes t_1 y t_2 entre sí, es decir, que se tienen nodos distintos a_1 y a_2 tales que t_1 es hijo de a_1 y t_2 es hijo de a_2. Supongamos que *nivel*(a_1)≥*nivel*(a_2) (en otro caso el razonamiento es igual). Entonces podemos intercambiar los hijos actuales de ai y as de forma que en a, queden como hijos t1 y t2 sin que la suma ponderada de la frontera de *B* se vea incrementada, ya que habremos subido de nivel lotes de mayor tamaño. Así vemos que en efecto los lotes t1 y t2 se mezclan en un árbol óptimo.

Para una implementación eficiente del algoritmo utilizamos un montículo de mínimos (véase el Capítulo 8). Los elementos del montículo serán los árboles de mezclas que se van obteniendo al mezclar los lotes y la prioridad correspondiente a cada árbol es el valor de su raíz. Inicialmente cada lote da lugar a un árbol con una hoja que contiene el tamaño del lote. En cada etapa se extraen del montículo los dos árboles con menor valor en la raíz y se crea un árbol

de mezclas con estos dos como hijos y la suma de sus tamaños en la raíz. Este nuevo árbol se añade al montículo. Tras *n*-1 etapas ya solo queda un árbol de mezclas en el montículo, que es el resultado buscado.

```
fun mezclaOptima(T[1..n] de nat+) dev
mezcla: árbol-bin[nat+]
var M: monticulo[árbol-bin[nat+]], a, m1, m2:
árbol-bin[nat+]

M:= montículo-vacío();
para i=1 hasta n hacer
    a:= plantar(árbolVacío(), T[i]J. árbolVacío());
    añadir(M, a);
fpara

para i=1 hasta n-1 hacer
    m1:= mínimo(M);
    eliminarMín(M);
    m2:= mínimo(M);
    eliminarMín(M);
    a:= plantar(m1, raíz(m1) + raíz(m2),m2)
    añadir(M, a);
fpara
mezcla:=mínimo(M) ;
eliminarMín(M);
ffun
```

Según se detalló en el Ejercicio 8.4. la obtención del elemento mínimo de un montículo tiene un coste constante, mientras que su eliminación y añadir elementos al montículo se realizan con un coste logarítmico en el tamaño del montículo. A partir del resultado obtenido en el Ejercicio 7.3 se deduce que el coste total del algoritmo está en $\Theta(n \log a)$.

12.13. Pavimentar calles

Los residentes de Barro City, debido a su gran tacañería, en vez de pavimentar todas las calles de la ciudad quieren pavimentar sólo las suficientes para poder ir de una intersección a otra cualquiera de la ciudad siguiendo una ruta pavimentada, y además, quieren gastarse tan poco dinero como sea posible en la realización de esta obra.

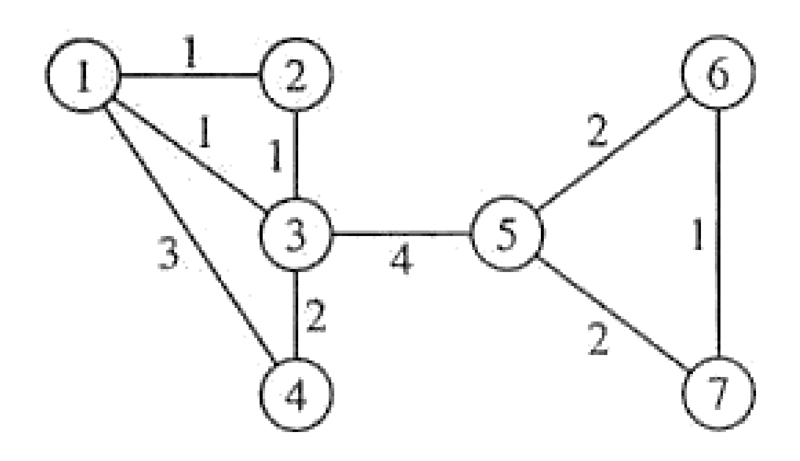
-----------------------------Solución--------------------------

Si consideramos las intersecciones de Barro City como vértices de un grafo. donde las aristas son las calles de la ciudad, etiquetadas con el coste correspondiente de pavimentación (valores reales positivos), lo que se obtiene es un *grafo valorado no dirigido* (se supone que las calles son de 2 direcciones) y conexo.

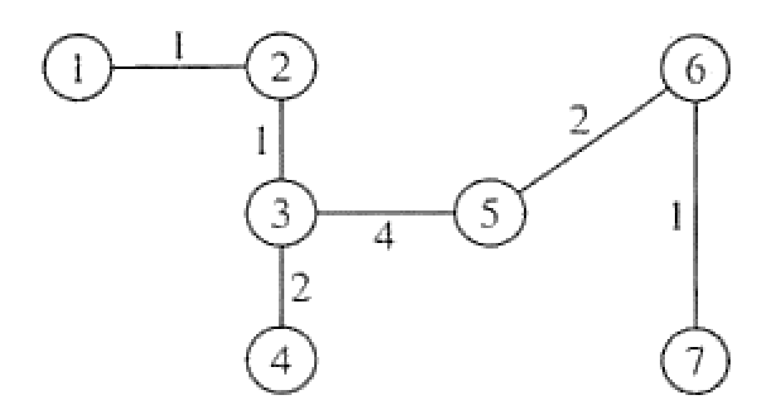
El objetivo es pavimentar las suficientes calles para poder acceder a todas las intersecciones desde cualquier otra, pero gastando lo mínimo en pavimentación. Eso significa encontrar un subgrato que contenga a todos los vértices, que siga siendo conexo y que el coste total de sus aristas sea mínimo: es decir, se quiere encontrar un ***árbol*** *(libre)* ***de recubrimiento de coste mínimo*** (si el subgrato contiene un ciclo, se puede eliminar una arista de dicho ciclo sin perder la propiedad de ser conexo y logrando un coste total menor). En la Figura 12.6 se muestra un grafo no dirigido valorado y conexo, y uno de sus

árboles de recubrimiento de coste mínimo. La idea para construir un árbol de recubrimiento se refleja en el siguiente esquema:

```
AR := ∅ ;
candidatas := A
mientras queden vértices sin conectar
hacer
   a := seleccionar(candidatas);
   candidatas := candidatas - {a};
   si sin-ciclos?(AR U {a}) entonces
     AR := AR U {a};
   fsi
fmientras
```



(a) Grafo no dirigido valorado y conexo



(b) Árbol de recubrimiento de coste mínimo

Figura 12.6: Ejemplo de árbol de recubrimiento de coste mínimo.

Dado un grafo valorado conexo, cuyas aristas tienen asociados valores numéricos no negativos, se dice que un subconjunto de aristas *T* es ***prometedor***

si se puede extender (añadiendo aristas) para obtener un árbol de recubrimiento de coste mínimo. El conjunto vacío siempre es prometedor (pues todo grafo conexo contiene al menos un subgrafo acíclico conexo que alcanza a todos los vértices), así que para que el árbol de recubrimiento finalmente obtenido sea de coste mínimo, es necesario definir una *función de selección* de aristas adecuada que mantenga *AR* prometedor en cada etapa. Presentamos aquí 2 estrategias diferentes que se corresponden con algoritmos bien conocidos:

- Escoger la arista de *menor valor* entre las restantes, manteniendo **conexo el subgrafo** que se está construyendo *(algoritmo de **Prim**).*
- Escoger la arista de *menor valor* entre las restantes, aunque el subgrafo resultante no sea conexo *(algoritmo de **Kruskal**).*

El algoritmo de Prim parte de un vértice cualquiera y va extendiendo el árbol de recubrimiento, incorporando en cada etapa un nuevo vértice.

En cambio, el algoritmo de Kruskal parte de un *bosque* de árboles formados por un único vértice cada uno, y en cada etapa del algoritmo se conectan un par de árboles mediante una arista, hasta que en el bosque quede un único árbol.

La corrección de ambos algoritmos se basa en el siguiente resultado:

Lema:

Sea $G = \langle V, A \rangle$ *un* grafo valorado conexo

con aristas de valores numéricos no negativos. Sean $W \subset V$, $T \subset A$ prometedor tal que si $\langle v,w \rangle \in T$ entonces $v,w \in W$ o $v,w \notin W$, y sea $a = \langle x, y \rangle \in A$ una de las aristas con menor valor con x e∈ W e y ∉ W. Entonces $T \cup \{a\}$ es prometedor.

Demostración:

Sea $R = \langle V, B \rangle$ un árbol de recubrimiento de coste mínimo de *G* con $T \subset B$ (*R* existe porque *T* es prometedor).

Si $a \in B$ entonces ya tenemos que $T \cup \{a\}$ es prometedor.

Si $a \notin B$, se tiene que en $\langle V, B \cup \{a\} \rangle$ habrá un ciclo y formando parte de ese ciclo habrá una arista $b = \langle v, w \rangle \in B$ con w ∈ IV A *v* ∉ W, como ilustra el dibujo de la Figura 12.7. Por las hipótesis del lema se tiene que *b* ∉ *T* y valor(a, G) ≤ valor(b, G), por lo que T U (a) C *C* = *(B* - {¿))U{a) y, además, (V. *C)* es también un árbol de recubrimiento de G y de coste mínimo, puesto que *R* lo era y hemos sustituido una arista por otra de valor no mayor.

Hemos explicado que el algoritmo de Prim comienza con un vértice cualquiera y va "conectando" vértices en cada etapa. Si $W \subset V$ es el conjunto de vértices ya conectados y *ARM* es el conjunto de aristas seleccionadas por el algoritmo de Prim, se cumple que todas las aristas en *ARM* tienen su origen y su destino en W. Supongamos que al comienzo de la etapa í-ésima *ARM* es prometedor y que se selecciona la arista $a = \langle x,y \rangle \in A$, eso significa que $x \in W \wedge y \notin W$ y que el coste

de *a* es el menor de entre todas las aristas en *A* que tienen su origen en W y su destino en *V* - W. Así pues, se cumplen todas las hipótesis del lema, por lo que *ARM* U {a} sigue siendo prometedor (W pasa a ser W U {y}). El algoritmo termina cuando todos los vértices están conectados, es decir, cuando W=V; entonces *ARM* no solo es prometedor, sino que *⟨V, ARM⟩* es un *árbol de recubrimiento de coste mínimo* del grafo de partida.

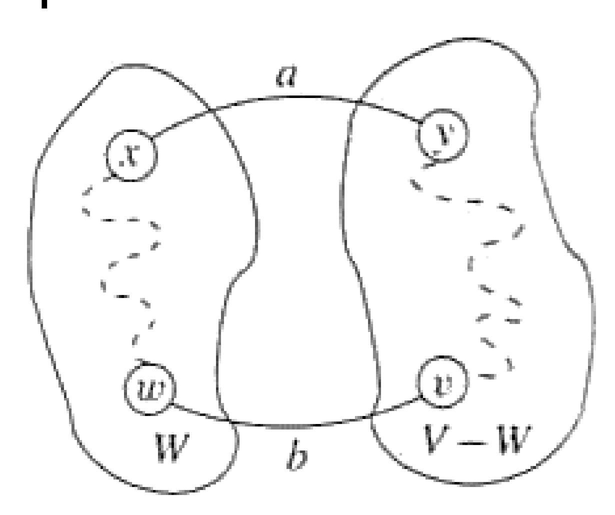


Figura 12.7: Selección de aristas para árbol de recubrimiento.

Veamos ahora la corrección del algoritmo de Kruskal. Sea *ARM* el conjunto de aristas escogidas para la solución por el algoritmo antes de comenzar la etapa í-ésima y que forman un "bosque de recubrimiento" (un conjunto de árboles libres que recubren todo el conjunto de vértices del grafo); en dicha etapa se selecciona una arista *a* = ⟨x, y⟩ ∈ A. que solo es finalmente incorporada a la solución si no forma ciclos con las demás aristas en *ARM.* Eso significa que x e *y* no pueden pertenecer al mismo árbol. Si denominamos IV el subconjunto de vértices conectados

(mediante un árbol) al que pertenece x, entonces se tiene que en *ARM* no hay aristas con el origen en W y el destino en V – W, y que *a* tiene el menor coste de entre todas las aristas en A que tienen su origen en W y su destino en V - W. De nuevo se cumplen todas las hipótesis del lema, por lo que *ARM* U {a} sigue siendo prometedor. El algoritmo termina cuando el bosque contiene un único árbol; entonces *ARM* corresponde a un árbol de recubrimiento y, como ARM es prometedor, ⟨V, *ARM*⟩ es un *árbol de recubrimiento de coste mínimo* del grafo de partida.

En los Ejercicios 12.14 y 12.15 veremos diferentes implementaciones del algoritmo de Prim; en el Ejercicio 12.16 se dará una implementación del algoritmo de Kruskal.

12.14. Prim

Implementar el algoritmo de Prim (véase el Ejercicio 12.13) que obtiene un árbol de recubrimiento de coste mínimo para un grafo no dirigido conexo cuyas aristas tienen asociados valores numéricos no negativos, suponiendo que el grafo viene representado mediante su matriz de valores (véase el Ejercicio 9.6).

------------------------Solución:-----------------------

Recordemos que el algoritmo de Prim comienza con un vértice cualquiera y en cada etapa conecta el vértice que menos le cueste. Para cada vértice que queda por incorporar al árbol que va construyendo el algoritmo, llevaremos calculado el coste mínimo para conectarlo al árbol *(costeMín)* y el vértice del árbol que permite la conexión *(conexión).* De esta forma, bastará encontrar el mínimo de entre los valores de *coste-mín;* la correspondiente *conexión* nos permitirá determinar la arista a incorporar al árbol.

Los valores de *costeMín y conexión* deben actualizarse para los vértices que todavía no se han incorporado al árbol; para ello, se comprueba si el último vértice incorporado, sea r. permite reducir el coste de conexión a los otros vértices; es decir, si para algún w se tiene que gv-valor(v, w, G) < *costeMin*(w): en caso afirmativo, el valor de *conexión(w)* pasaría a ser v. Para facilitar estas comprobaciones, elegimos +∞ como valor especial en la matriz de valores para las aristas que no existen en

el grafo. Extendemos las operaciones aritméticas para operar con valores infinitos de la forma esperada.

Si suponemos que los vértices del grafo están fijados y representados con naturales 1*..n.* podemos utilizar la implementación para grafos valorados dada en el Ejercicio 9.6 y representar *costeMín* y *conexión* mediante un par de vectores. Si el algoritmo comienza incorporando el vértice 1 al árbol, para el resto de vértices del grafo los valores de *costeMin*[i] y *conexión[i*] se inicializarán con el valor de *gv-valor*(1, i, G) y 1 respectivamente.

Para que los vértices ya incorporados al árbol no interfieran en la búsqueda del mínimo ni en la actualización de los costes mínimos, su valor de *costeMín* será -1.

Siguiendo todas estas ideas, el algoritmo resultante es el siguiente:

```
tipos
   arista = reg
            origen : 1..n
            destino : 1..n
          freg
ftipos
{G es conexo}
fun Prim1(G : grafo-val[n]) dev ARM :
```

```
conjunto[arista]
var costeMin[2..n] de real∞.,
      conexión[2..n] de 1..n,
      a: arista

ARM := cjto-vacío()
para i = 2 hasta n hacer
  costeMin[i] := G[1,i];
  conexión[i] := 1;
fpara

para i = 1 hasta n - 1 hacer
{encontrar el mínimo en coste-mín}
  mínimo := +∞;
   para j = 2 hasta n hacer
    si 0 ≤ costeMin[j] ∧ costeMin[j] <
    mínimo entonces
      mínimo := costeMin[j] ;
      elegido := j;
    fsi
   fpara
   a.origen := elegido;
   a.destino := conexión[elegido];
   añadir(a, ARM);
   costeMin[elegido := -1;
   para j=2 hasta n hacer {actualizar los
   vectores}
    si G(elegido, j] < costeMin[j]
    entonces
      costeMin[j] := G[elegido, j];
      conexión[j] := elegido;
    fsi
   fpara
fpara
ffun
```

El COSTE en tiempo de esta implementación está en Θ(n^2), pues se

realizan *n*-1 etapas y en cada etapa se realizan en secuencia dos bucles de *n*-1 iteraciones cada uno. Como se ve. el coste es independiente del número de aristas en el grafo. En el Ejercicio 12.15 se presenta otra implementación del algoritmo de Prim, con un coste en *Q(m* logn), siendo *m* el número de aristas.

El coste en espacio adicional está en Θ(n), por los 2 vectores auxiliares, *coste-mín* y *conexión.*

12.15. Prim

Implementar el algoritmo de Prim (ver Ejercicio 12.13) que obtiene un árbol de recubrimiento de coste mínimo para un grafo no dirigido conexo cuyas aristas tienen asociados valores numéricos no negativos, suponiendo que el grafo viene representado mediante listas de adyacentes (ver Ejercicio 9.6), y utilizando un montículo (ver Capítulo 8) como estructura de datos para representar los candidatos pendientes.

------------------------Solución--------------------------

El algoritmo de Prim comienza con un vértice cualquiera y en cada etapa conecta el vértice que menos cueste. En la implementación dada en el Ejercicio 12.14 se utilizan dos vectores *(coste-mín* y *conexión)* para llevar para cada vértice el coste mínimo de conexión al árbol que se está construyendo y el vértice que permite dicha conexión. Pero la búsqueda del mínimo en el vector *coste-mín* requiere un coste lineal en el número de vértices, lo que conduce a un coste total cuadrático. Para intentar reducir el coste en tiempo, recurrimos a una estructura que proporciona acceso al mínimo elemento en un tiempo constante: un montículo de mínimos, donde los elementos serán los vértices y la prioridad asociada será el coste mínimo de conexión.

Ahora bien, si se quiere sacar provecho de la reducción en tiempo obtenida con el uso del montículo, no interesa que el

grafo venga dado por la matriz de valores, porque en ese caso la búsqueda de los vértices adyacentes al vértice seleccionado sería lineal en *n* y el coste total seria otra vez cuadrático. Por eso se utilizan las listas de adyacentes, que permiten recorrer exclusivamente las aristas existentes en el grafo. Al recorrer la lista de adyacentes del vértice seleccionado e ir modificando el coste mínimo de los vértices adyacentes, debemos ser capaces de localizar eficientemente dichos vértices adyacentes en el montículo, para modificar su prioridad. Para ello utilizaremos los *montículos de pares* implementados en el Ejercicio 8.8. los cuales admiten ese tipo de modificación con un coste reducido (logarítmico respecto al número de elementos en el montículo).

Supondremos, una vez más, que los vértices del grafo están fijados y representados con naturales 1 *,n.* Llevaremos los vectores *coste-mín* y *conexión* como en la versión del Ejercicio 12.14. Primero inicializamos *coste-mín* con el valor $+\infty$ para todos los vértices; a continuación se recorre la lista de adyacentes del primer vértice, para actualizar los adyacentes con el valor de la arista que llega desde el primer vértice. Esos vértices adyacentes además son incluidos en el montículo. En todo momento, el montículo contiene solamente vértices que todavía no han

sido seleccionados pero cuyo coste de conexión al árbol no es +∞, es decir, que son accesibles desde alguno de los vértices del árbol construido. Para que los vértices ya incorporados al árbol no den problemas al recorrer las listas de adyacentes, su valor de *coste-mín* se pondrá a -1, incluido el del primer vértice.

Los montículos de pares implementados en el Ejercicio 8.8 están definidos para valores que son números naturales en el intervalo *\..N.* En el caso presente, los valores son los *n* vértices del grato, por lo que instanciaremos *N* con el valor de *n.*

```
tipos
    arista = reg
        origen: 1..n
        destino:1..n
      freg
ftipos
{ G es conexo}
```

```
fun Prim2(G : grafo-val[n]) dev ARM :
conjunto[arista]
var coste-min[1..n] de real∞,
    conexión[2..n] de 1..n,
    M: montículo-pares,
    mínimo: par,
    a: arista,
    l: lista[info-arista]

ARM := cjto-vacío();
M := montículo-vacío();
coste-min[1] := -1;
coste-min[2..n] := [+∞];
```

```
conexión[2..n] := [1];
{actualizar el coste mínimo para los adyacentes al
primer vértice
l := copiar-lista(G[1]);
  mientras ¬ es-lista-vacía?(l) hacer
    i := izquierdo(l).destino;
    coste-mín[i] := izquierdo(l).valor;
    anadir(M, i, coste-min[i]);
    elim-izq(l);
  fmientras

  para i = 1 hasta n-1 hacer
    mínimo := mínimo(M);
    eliminar-mín(M);
    elegido := mínimo.elem;
    a.origen := elegido;
    a.destino := conexión[elegido];
    añadir(a, ARM);
    coste-mín[elegido] := -1;
    l := copiar-lista(G[elegido]);
    mientras ¬ es-lista-vacía?(l) hacer
      i := izquierdo(l).destino;
      si izquierdo(l).valor < coste-min[i]
      entonces
        coste-mín[i] := izquierdo(l).valor;
        conexíón[i] := elegido;
        modificar(M, i, coste-mín[i]); {actualizar
        la prioridad del vértice adyacente}
      fsi
      elim-izq(l);
    fmientras
  fpara
ffun
```

La inicialización de este algoritmo tiene un coste en Θ(n logn) ya que la lista de adyacentes del primer vértice tiene una *longitud* máxima de *n* y añadir cada par en el montículo tiene coste logarítmico.

El bucle principal del algoritmo realiza *n* - 1 etapas; en cada una de ellas se

obtiene el mínimo del montículo (coste constante) y se elimina dicho mínimo (coste logarítmico); además, en el conjunto de las n - 1 etapas, se accede a todas las aristas del grafo (con coste constante cada acceso) y por cada una de ellas, en el caso peor, se realiza una modificación en el montículo (coste logarítmico).

Resumiendo, el coste total en tiempo del algoritmo está en $\Theta(n\ \mathit{logn} + m \log n) = \Theta(m \log n)$, siendo m el número de aristas, porque al ser un grafo conexo, se tiene que $n - 1 \leq m \leq n^2$. Comparando con el algoritmo Prim1 del Ejercicio 12.14. se observa que el uso del montículo solamente merece la pena si el número de aristas es pequeño y cercano a n (el grafo es disperso), porque si se acerca a $n2$ (el grado es denso), el algoritmo Prim2 tiene un coste en $\Theta(n^2 \log n)$ que es peor que el coste de Prim1 que siempre está en $\Theta(n^2)$ y es independiente de m.

El *coste* en espacio adicional corresponde al del montículo de pares y los dos vectores, así que está en $\Theta(n)$.

12.16. Kruskal

Implementar el algoritmo de Kruskal (véase el Ejercicio 12.13) que obtiene un árbol de recubrimiento de coste mínimo para un grafo no dirigido conexo cuyas aristas tienen asociados valores numéricos no negativos, suponiendo que el grafo viene representado mediante su matriz de valores (véase el Ejercicio 9.6).

-------------------------Solución------------------------

Recordemos que el algoritmo de Kruskal parte de un *bosque* de árboles formados por un único vértice cada uno, y en cada etapa del algoritmo se conectan un par de *árboles* mediante una arista, hasta que en el bosque solamente quede un árbol. Esta visión del algoritmo nos sugiere la utilización de las estructuras de partición especificadas en el Ejercicio 9.16. Si consideramos como relación de equivalencia entre los vértices de un grafo el pertenecer al mismo árbol, tendremos una relación de equivalencia dinámica que puede representarse mediante una partición de los vértices. Al principio cada vértice está aislado y a medida que se van añadiendo aristas, se “fusionan" los árboles.

El algoritmo de Kruskal selecciona en cada etapa la arista de menor coste, siempre que no forme ciclos con el conjunto de aristas previamente seleccionadas: para asegurar esto basta comprobar que sus extremos pertenecen a árboles (clases de equivalencia) distintos.

La operación aristas (especificada e implementada en el Ejercicio 9.7) obtiene el conjunto de aristas de un grafo (de hecho una lista en la implementación). pero para facilitar la selección de las aristas es conveniente que estas estén ordenadas por coste *creciente* en alguna estructura que no sea un conjunto, por ejemplo, un vector. En una primera fase se introducen en el vector todas las aristas junto con su coste asociado, y en una segunda fase se ordena el vector.

Suponiendo que los vértices del grafo están fijados y representados con los números naturales 1..n, podemos utilizar la última implementación de las estructuras de partición dada en el Ejercicio 9.17. El algoritmo queda así:

```
tipos
    arista-val = reg
            origen : 1..n
            destino : 1..n
            valor : real∞
          freg
  ftipos
{G es conexo}
fun Kruskal (G : grafo-val[n]) dev ARM :
conjunto[arista-val]
var P : partición[1..n],
      A[1..zz²] de arista-val,
      a : arista-val

ARM := cjto-vacío();
núm-aristas := 0
{núm-aristas indica el cardinal del conjunto
ARM}
P := crear-partición3()
```

```
⟨A, m⟩ := obtener-vector-aristas(G) {aristas
en A[1..m]}
ordenar(A, m) {por costes crecientes}
i := 1
mientras núm-aristas < n-1 hacer
    a ;= A[i];
    i := i +1 {seleccionar arista siguiente}
    c := buscar3(a.origen, P) ;
    d := buscar3(a.destino, P)
    si c ≠ d entonces
        fusionar3(c, d, P)
        añadir(a, ARM) ;
        núm-aristas := núm-aristas + 1
    fsi
fmientras
ffun
```

No detallamos aquí la obtención del vector 4 con las aristas de G (el algoritmo obtener-vector-aristas). porque es una modificación sencilla de los algoritmos dados en el Ejercicio 9.7. Interesa que las aristas incluidas en el vector sean las que tienen el primer *extremo* estrictamente menor que el segundo (aunque pudieran existir auto-aristas en el grafo, no tienen ningún interés, pues nunca van a ser seleccionadas porque forman ciclo). El coste de esta fase está en Θ(n2) para la matriz de valores y en Θ(n+m) para las listas de adyacentes.

El coste de ordenar el vector de aristas está en Θ(m log m), o equivalentemente en Θ(m log n) ya que n-1 ≤ m ≤ n(n-1)/2.

Crear la partición tiene un coste en Θ(n). En cuanto al *coste* del bucle, se observa que en el peor de los casos,

cuando se consideran todas las aristas del grafo. se realizan m iteraciones, con 2m búsquedas en la partición *y* *n*-1 fusiones (hasta obtener un único árbol en el bosque o, lo que es lo mismo, un único conjunto en la partición). Como ya se detalló en el Ejercicio 9.17. la representación aquí utilizada garantiza que el coste de estas *c*=2m + (n-1) operaciones es prácticamente lineal en c. De esta forma, el coste total del algoritmo puede considerarse del orden del coste de la tase de inicialización (antes del bucle), que tiene un coste que está en 0(m log n) para las listas de adyacentes y en 0(n2) para la matriz de valores.

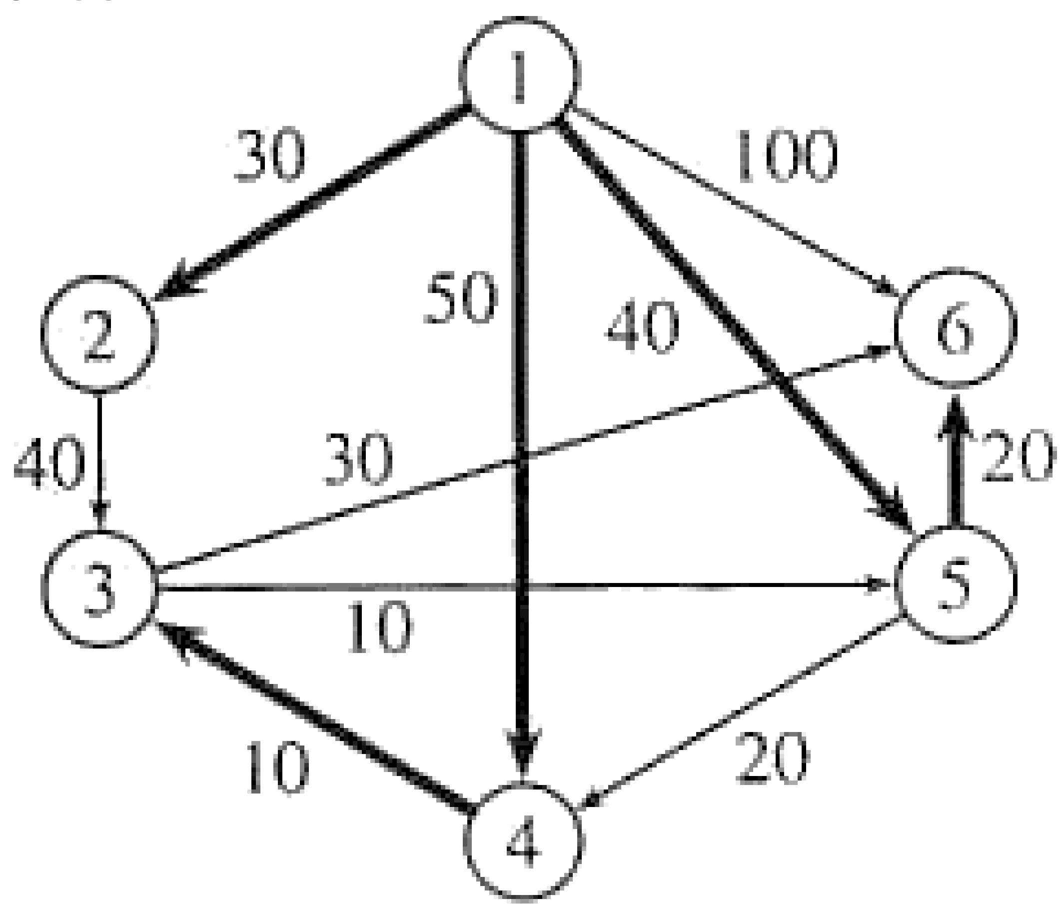


Figura 12.8: Ejemplo de caminos de coste mínimo.

Estos costes son similares a los de las versiones dadas para el algoritmo de Prim: Θ(m log *n*) para las listas de adyacentes y montículo (ver Ejercicio

12.15) y $\Theta(n^2)$ para la matriz de valores y buscando el mínimo en el vector (ver Ejercicio 12.14).

12.17. Fantasma

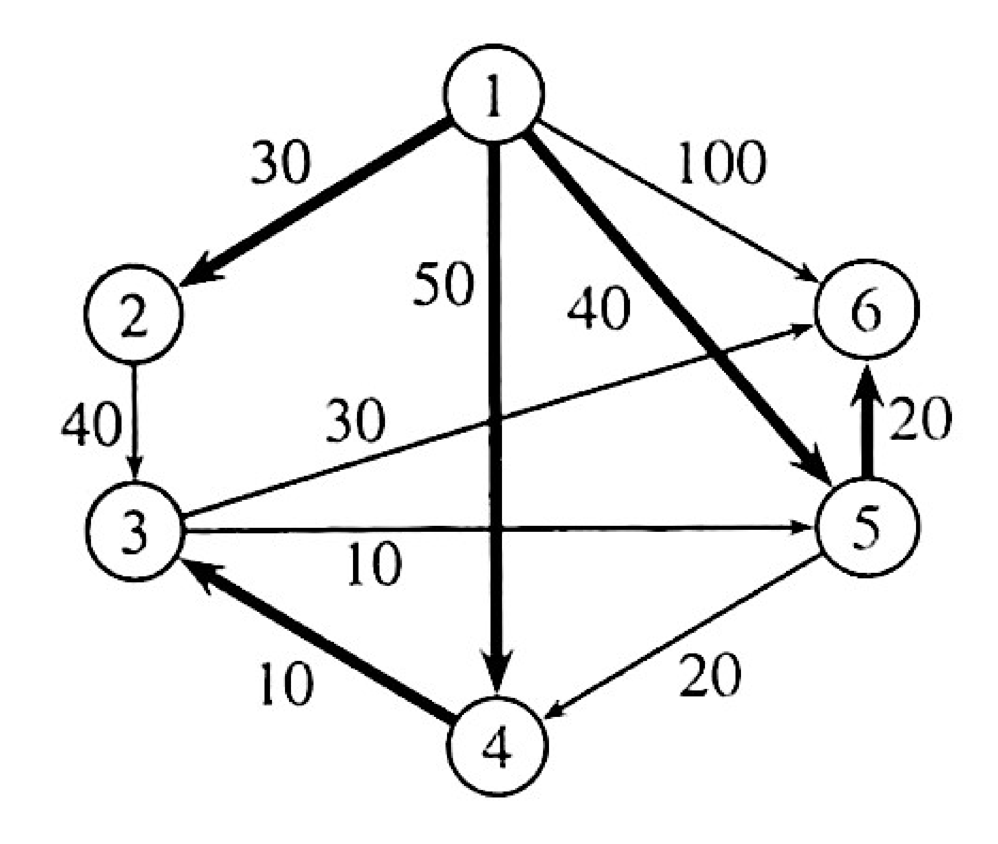
El fantasma de la zarzuela vive oculto en las salas más recónditas del viejo Teatro de la Zarzuela. El fantasma conoce la existencia de una olvidada red de túneles que recorren el subsuelo de la ciudad, y lo utiliza para desplazarse en sus correrías nocturnas sin ser descubierto *y* sin enfangarse. Por los túneles transcurre una corriente subterránea, de forma que hay que desplazarse por ellos mediante una barca que es arrastrada en el sentido de la corriente. La red tiene una entrada secreta desde el teatro y diversas salidas en puntos estratégicos de la ciudad.

El fantasma desea conocer el camino más corto para desplazarse por la red subterránea desde el teatro a los otros puntos estratégicos.

-------------------------Solución------------------------

Si consideramos las entradas y salidas de la red subterránea como vértices de un grafo, donde las aristas son los túneles que las conectan, etiquetadas con su correspondiente longitud (valores reales positivos), lo que se obtiene es un *grafo valorado dirigido* en el que todos los vértices son accesibles desde un determinado vértice (la entrada del teatro). El problema a resolver consiste entonces en determinar el ***camino de coste mínimo*** desde ese determinado vértice hasta cada uno de los demás vértices del grafo. Por ejemplo, en el grafo de la Figura 12.8 se muestran en trazo más

grueso los caminos de coste mínimo desde el vértice 1 hasta los demás vértices.



Una manera de resolver este problema es mediante el *algoritmo de* Dijkstra, que consiste en realizar *n*-1 etapas (para un grafo de *n* vértices), en cada una de las cuales se fija el camino de coste mínimo desde el vértice origen a uno de los vértices.

Durante la ejecución del algoritmo, el conjunto de vértices queda repartido en 2 subconjuntos: el de los ***seleccionados***, para los que ya se conoce el coste mínimo desde el origen, y el de los ***no seleccionados***, para los cuales se conoce el coste del camino mínimo desde el origen, pero pudiendo atravesar solamente vértices seleccionados *(camino interno).* Inicialmente el único vértice seleccionado es el origen; en cada etapa se selecciona el vértice que en ese momento tenga el *menor coste desde el origen*, y dicho vértice seleccionado sirve

para actualizar el coste desde el origen hacia los vértices restantes, porque los caminos internos ya podrán pasar por dicho vértice. El algoritmo termina cuando los *n* vértices han sido seleccionados. En este caso no se necesita un test de factibilidad porque todos los candidatos son admisibles.

Sea el origen el primer vértice, v_1. Para el resto de los vértices del grafo *coste-mín* se inicializa con el valor de la arista que llega desde v_1 si es que esta existe: en caso contrario se da el valor $+\infty$. que es mayor que cualquier posible coste.

El algoritmo queda de la forma que viene a continuación. Para las operaciones que quedan sin detallar referimos a las implementaciones más concretas en los Ejercicios 12.18 y 12.19.

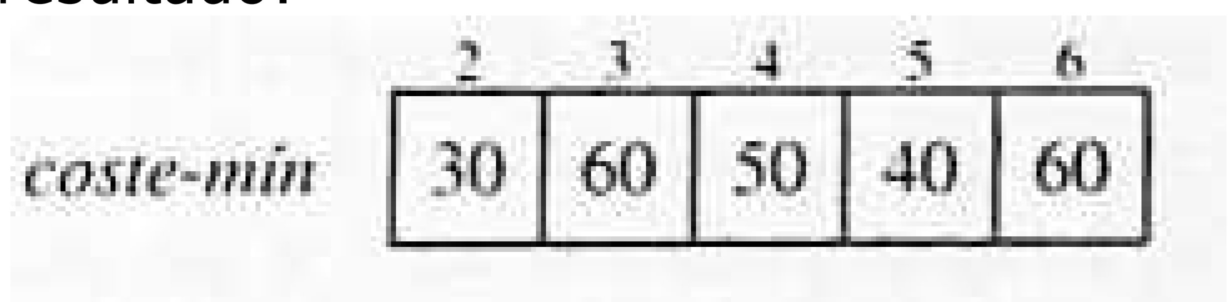
{en *G* todos los vértices son accesibles desde v1)

```
fun Dijkstra(G: grafo-val[n], v1: 1..n) dev
coste-mín[1..n] de real∞
var candidatos: conjunto[1..n], l: lista[1..n]

candidatos := cjto-vacío();
para i = 1 hasta n hacer
   añadir(i, candidatos);
fpara
quitar(v1, candidatos);
coste-min[1..n] := [+∞];
coste-min[v1] := 0;
l := adyacentes(v1, G);
mientras ¬ es-lista-vacía?(l) hacer
   w := izquierdo(l);
   elim-izq(l);
   coste-min[v] := gv-valor(v1, v, G);
fmientras
```

```
para i = 1 hasta n - 2 hacer
    v := mínimo(candidatos, coste-mín) {
    determinar el candidato con menor coste-mín}
    quitar(v, candidatos);
    l := adyacentes(v, G);
    mientras ¬ es-lista-vacía?(l) hacer
        w := izquierdo(l);
        elim-izq(l);
        coste-min[w] := mín(coste-min[w], coste-
        min[v] + gv-valor(v, w, G));
    fmientras
fpara
ffun
```

Si aplicamos este algoritmo al grafo de la Figura 12.8 obtendremos el siguiente resultado:



	2	3	4	5	6
coste-mín	30	60	50	40	60

Nótese la similitud de este algoritmo con el de Prim para obtener árboles de recubrimiento de coste mínimo (véanse los Ejercicios 12.14 y 12.15). Al igual que en aquel caso, presentaremos dos implemen- taciones diferentes del algoritmo de Dijkstra: una con matriz de valores y otra que utiliza un montículo para representar los candidatos pendientes (véanse los Ejercicios 12.1S y 12.19. respectivamente).

Obsérvese que el algoritmo tan solo realiza *n*-2 etapas, es decir, que el último candidato no llega a ser seleccionado nunca. Sin embargo, esto no resulta necesario porque esa última etapa

adicional ya no modificaría los costes mínimos de ningún vértice del grato.

Tal como está, el algoritmo solamente devuelve los costes de los caminos mínimos; si se desea obtener información sobre los caminos que ofrecen esos costes mínimos, es necesario devolver además un vector que recoja el vértice predecesor en un camino de coste mínimo. Un vector resulta suficiente, porque cada camino de coste mínimo está formado por subcaminos de coste mínimo.

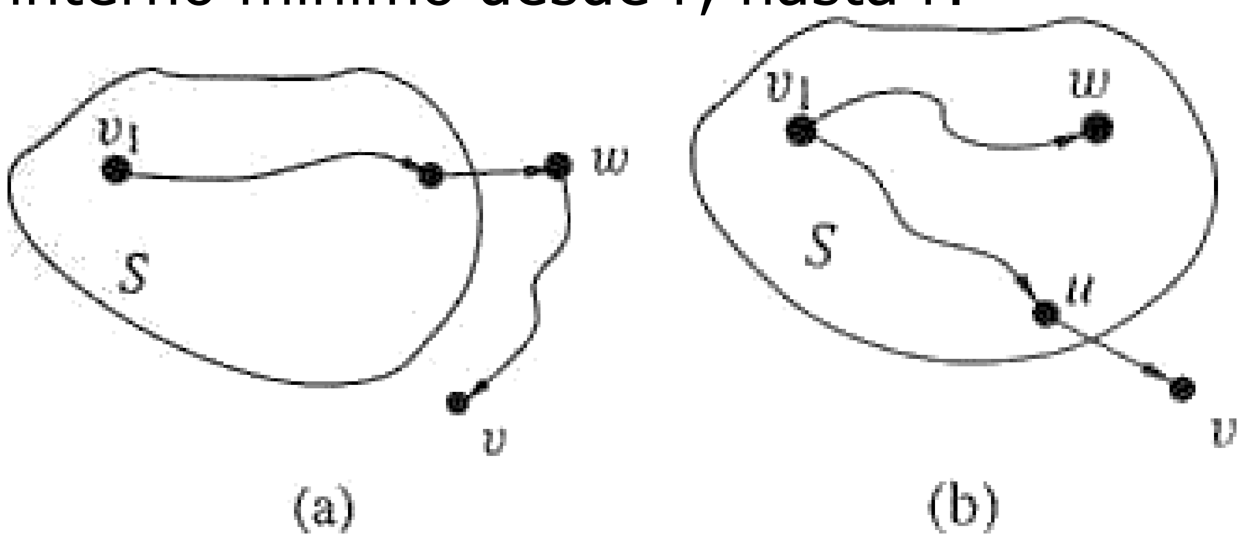
La corrección del algoritmo de Dijkstra se basa en el siguiente resultado:

Lema:

Sea *G* = ⟨V, A⟩ un grafo dirigido con aristas valoradas con números no negativos, y sea S = *V - candidatos.* Entonces, en cada etapa del algoritmo de Dijkstra se cumple que:

1. Vv 6 S — {i'i) : c<wk'-nn7i[v] = T(i'|. t>); y

Vi> e *candidatos :* = coste del camino interno mínimo desde r, hasta r.



(a) (b)

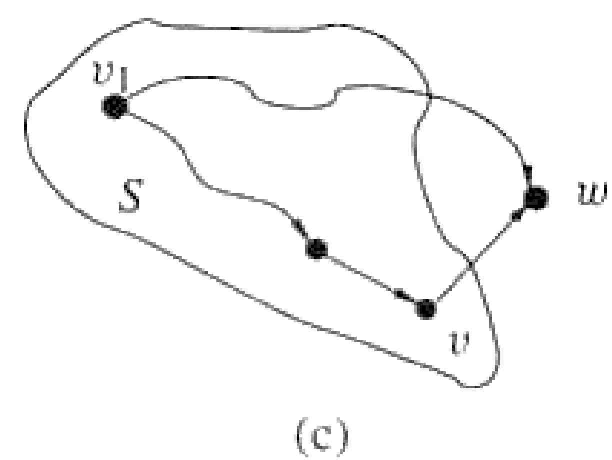


Figura 12.9: Demostración del algoritmo de Dijkstra.

Demostración:

Se demuestran 1 y 2 simultáneamente por inducción sobre el número de etapa. En la primera etapa se tiene S = [ni}, por lo que el primer resultado es trivialmente cierto (no se aplica a ningún vértice), y el segundo resultado se verifica porque en este caso los caminos internos corresponden a las aristas con origen en vi y *coste-mín* se inicializa con el valor de estas aristas. Supongamos que en la etapa í-ésima se ha seleccionado un vértice u con u e *candidatos.* Como el primer resultado se satisface para la etapa anterior, para los vértices en *S coste-mín* no puede cambiar al añadir *v.* En cuanto al propio vértice u se tiene lo siguiente: consideremos un camino de coste mínimo desde v, hasta *v, y* sea *w* el primer vértice que pertenece al camino y no está en S, como se indica en el dibujo de la Figura 12.9(a). Entonces el camino que va desde u, hasta *w* es interno, y debe tener el menor coste. Así que F(ui,u) = Tít'i, tu) + F(UÍ. u) — cos/e-niñi[u>] + r(u>, v) > cosre-

/núi[u] + F(u;. t>) > cosze-znm[u].
Pero por definición de T se tiene que r(m, u) < cotfe-míhfv], por lo que se concluye que F(i>i, v) = co.sz<?-mzn[v].

Nos queda por demostrar el segundo resultado. Para ello necesitamos demostrar primero el siguiente resultado auxiliar:

Vtu e 5 : F(ui, tu) < r(vi, v),

que demostramos también por inducción sobre el número de etapa.

En la primera etapa el resultado se cumple trivialmente porque S contiene únicamente a V|. Sea ahora la etapa en la que se selecciona el vértice *v* y consideremos un camino (interno) de coste mínimo desde hasta u. y sea *u* el vértice que precede a *v* en ese camino, tal y como se indica en el dibujo de la Figura 12.9(b). Tenemos entonces dos posibilidades:

- íí se incorporó a S en una etapa anterior que *w; eso* significa que al seleccionar *w* para ser añadido a S. *v* ya era accesible desde v, a través de íí, y si u no fue seleccionado en lugar de *w* es porque en aquel momento se tenía co.sm-/m'/i|w] < co.síe-mí>i[v], pero además se cumple que cosz<’-zníti[iu] = T(ui, tu) por el primer resultado del lema, y que cosZe-/ní)i[v] = T(U], u) porque el camino de coste mínimo para u ya era interno en aquella etapa.

- *u* se incorporó a S en una etapa posterior que *w;* entonces, por hipótesis de inducción de este resultado auxiliarse tiene que r(vj. *w)* < F(tq. íí) y además F(ui. tz) < F(ui. v).

Volviendo al segundo resultado del lema, debemos comprobar que el algoritmo tiene en cuenta correctamente todos los nuevos caminos internos, es decir, los que contienen al vértice *v* recién incorporado a S. De nuevo se tienen dos posibilidades:

- v es el último vértice interno en un camino desde vi hacia *w* (como muestra la Figura 12.9(c)): se cumple entonces que el coste de dicho camino es cojZe-/ztz'zz[v] + gv-valor(v, *w. G).* que es precisamente con lo que se compara cosZe-zzííh[iu] para ser actualizado si fuera necesario.
- v es un vértice interno en un camino interno desde vi hacia *w:* sea íí el último vértice interno en ese camino, entonces el resultado auxiliar nos dice queF(ui,«) < T(U|, *u),* por lo que el nuevo camino resultará al menos tan costoso como ir desde ui a *u* y de ahí directamente a *w.* Por eso no es necesario considerar ese tipo de caminos y el algoritmo no los tiene en cuenta.

12.18. Dijkstra

Implementar el algoritmo de Dijkstra (ver Ejercicio 12.17) para determinar los caminos de coste mínimo desde un origen fijo al resto de vértices en un grafo dirigido cuyas aristas están valoradas con números no negativos, y suponiendo que todos los vértices del grafo son accesibles desde el origen. Considerar que el grafo viene representado mediante su matriz de valores (véase Ejercicio 9.6).

----------------------Solución----------------------

En la función Dijkstra dada en el Ejercicio 12.17 no se detalló la forma de obtener el candidato con menor coste. La forma más sencilla es recorrer el vector *coste-mín.* Para que en dicho recorrido no se tengan en cuenta los valores correspondientes a los vértices ya seleccionados, ya no sirve el truco utilizado en el algoritmo de Prim de poner un valor especial (- 1) en *coste-mín* para esos vértices, puesto que esos costes mínimos forman parte del resultado de la función y no podemos perderlos. Lo que haremos será consultar si el vértice en cuestión pertenece al conjunto de candidatos pendientes. Para simplificar el algoritmo, supondremos que el origen es siempre el primer vértice 1; entonces los candidatos son valores entre 2 y *n* y es factible implementar el conjunto *candidatos* como un vector de valores booleanos. con lo que la consulta de pertenencia al conjunto tiene un coste

constante, así como las operaciones de añadir y quitar elementos del conjunto (véase el Ejercicio 2.2).

Además de los costes de los caminos mínimos, la función devuelve el predecesor de cada vértice en dicho camino.

De nuevo, elegimos +∞ como valor especial en la matriz de valores para las aristas que no existen en el grafo.

```
{en G todos los vértices son accesibles desde
el primer vértice}
fun Dijkstra1(G: grafo-val[n]) dev ⟨coste-
min[1..n] de valor, predecesor[2..n] de
1..n⟩
var candidatos: conjunto[2..n]

candidatos:= cjto-vacío()
coste-mín[1]:= 0
para i=2 hasta n hacer
   añadir(i, candidatos);
   coste-min[i]:= G[1,i];
   predecesor[i]:= 1;
fpara

para i=1 hasta n-2 hacer
   {encontrar el mínimo en coste-mín}
   mínimo:= +∞;
   para j=2 hasta n hacer
      si está?(j, candidatos) ∧ coste-mín[j] <
      mínimo entonces
         mínimo:= coste-mín[j];
         elegido := j;
      fsi
   fpara
   quitar(elegido, candidatos);
   para j=2 hasta n hacer {actualizar los
   vectores}
```

```
si coste-mín[elegido] + G[elegido, j] <
coste-min[j] entonces
    coste-mín[j]:= coste-mín[elegido] +
    G[elegido, j];
    predecesor[j]:= elegido;
fsi
fpara
fpara
ffun
```

El coste en tiempo de esta implementación está en Θ(n^2), puesto que se realizan *n*-2 etapas y en cada etapa se realizan en secuencia dos bucles de *n*-1 iteraciones cada uno. Como se ve. el coste es independiente del número de aristas en el grafo. En el Ejercicio 12.19 se presenta otra implementación del algoritmo de Dijkstra, con un coste en Θ(*m* log n), siendo *m* el número de aristas.

El coste en espacio adicional está en Θ(n) por el vector para implementar el conjunto de candidatos.

12.19. Dijkstra

Implementar el algoritmo de Dijkstra (ver Ejercicio 12.17) para determinar los caminos de coste **mínimo** desde un origen fijo al resto de vértices en un grafo dirigido cuyas aristas están valoradas con números no negativos, y suponiendo que todos los vértices del grafo son accesibles desde el origen. Considerar que el grafo viene representado mediante listas de adyacentes (ver Ejercicio 9.4), y utilizar un montículo (ver Capítulo 8) como estructura de datos para representar los candidatos pendientes.

------------------------Solución:------------------------

Si utilizamos un montículo de mínimos, donde los elementos son los vértices y la prioridad asociada es el coste mínimo dado por *coste-mín,* consultar el mínimo tiene coste constante y eliminarlo tiene coste logarítmico con respecto al número de vértices. Además, para evitar el coste en tiempo cuadrático en el recorrido de las aristas del grafo, se utilizan las listas de adyacentes, que permiten recorrer exclusivamente las aristas existentes en el grafo. Como ya se explicó para la versión con montículos del algoritmo de Prim (véase el Ejercicio 12.15). utilizaremos los montículos de pares implementados en el Ejercicio 8.8.

Comenzamos inicializando *coste-mín* con el valor +∞ para todos los vértices. A continuación se recorre la lista de adyacentes del primer vértice, para actualizar los adyacentes con el valor de la arista que llega desde el primer vértice.

El montículo contiene siempre todos los vértices que todavía no han sido

seleccionados y cuyo coste mínimo actual no es +oo. De esta forma no es necesario llevar de forma explícita el conjunto de candidatos, porque a los vértices ya seleccionados no se les modifica *coste-mín.*

Los montículos de pares implementados en el Ejercicio 8.8 están definidos para valores que son números naturales en el intervalo 1..AL En el caso presente, los valores son los *n* vértices del grafo, por lo que instanciaremos *N* con el valor de *n.*

```
{en G todos los vértices son accesibles desde el
primer vértice }
fun Dijkstra2(G : grafo-val[n]) dev ⟨coste-
min[1..n] de real∞, predecesor[2..n] de
1..n⟩
var M: montículo-pares, mínimo: par,
      l: lista[info-arista]

M:= montículo-vacío()
coste-min[1]:= 0
para i = 2 hasta n hacer
   coste-mín[i]:= +∞;
   predecesor[i]:= 1;
fpara

{actualizar el coste mínimo para los adyacentes
al primer vértice}
l:= copiar-lista(G[1])
mientras ¬ es-lista-vacía?(l) hacer
   i:= izquierdo(l).destino;
   coste-mín[i]:= izquierdo(l).valor;
   añadir(M, i, coste-min[i]);
   elim-izq(l);
fmientras

para i=1 hasta n-2 hacer
   mínimo:= mínimo(M);
```

```
    eliminar-mín(M);
    elegido:= mínimo.elem;
    l:= copiar-lista(G[elegido]);
    mientras ¬ es-lista-vacía?(l) hacer
      i:= izquierdo(l).destino;
      coste:= coste-mín[elegido] +
      izquierdo(l).valor;
      si coste < coste-mín[i] entonces
        coste-mín[i]:= coste;
        predecesor[i]:= elegido;
        modificar(M, i, coste); {actualizar la
        prioridad del vértice adyacente }
      fsi
      elim-izq(l);
    fmientras
  fpara
ffun
```

El análisis del COSTE (en tiempo y en espacio) de este algoritmo es muy similar al del algoritmo de Prim implementado con un montículo, visto en el Ejercicio 12.15. Allí se justificó que el coste en tiempo está en $\Theta(m \log n)$, siendo m el número de aristas, y en espacio adicional está en $\Theta(n)$.

De manera similar a lo que acontece al comparar las versiones del algoritmo de Prim. se observa que esta versión con el montículo merece la pena solo si el número de aristas es pequeño y cercano a *n* (grafo disperso); en cambio, si *m* se acerca a *ir* (grafo denso), entonces el algoritmo Dijkstra1 del Ejercicio 12.18 resulta menos costoso.

12.20. Dijkstra

Demostrar que los caminos construidos por el algoritmo de Dijkstra (ver Ejercicio 12.17) sobre un grafo *no dirigido* y conexo forman un árbol de recubrimiento. ¿Es un árbol de recubrimiento de coste mínimo? ¿Es el camino entre un par de vértices en un árbol de recubrimiento de coste mínimo necesariamente un camino de coste mínimo?

----------------------------Solución:--------------------------

Sea S el conjunto de vértices para los cuales se conoce el camino de coste mínimo. Inicialmente, S está formado exclusivamente por el vértice origen y por tanto es un árbol (conexo y acíclico). Supongamos que cuando el algoritmo de Dijkstra va a añadir el vértice *v* al conjunto S, los caminos de coste mínimo de los vértices en S forman un árbol. El camino de coste mínimo que llega hasta v es un camino interno que solo utiliza vértices de *S.* Si *w* es el vértice predecesor de v en este camino, la parte del camino de coste mínimo de *v* que va desde el origen hasta *w* coincide con el camino de coste mínimo para w; es decir, que al añadir *v* solamente se añade una arista, la que une w con v, la cual no forma ciclos, ya que *w* pertenecía a S *y v* no. Por tanto, el nuevo conjunto sigue formando un árbol. El algoritmo termina con todos los vértices del grafo seleccionados, por lo que se consigue un <u>árbol *de recubrimiento*</u>.

Sin embargo, este árbol de recubrimiento <u>NO</u> tiene necesariamente coste MÍNIMO. Por ejemplo, para el grafo de la Figura 12.10(a) y considerando el vértice 1 como

el origen, el algoritmo de Dijkstra tomaría los caminos mostrados en la Figura 12.10(b) con un coste total como árbol de 4. En cambio un árbol de recubrimiento se muestra en la Figura 12.10(c) con un coste total mínimo de 3.

El mismo ejemplo nos sirve para demostrar que el camino entre un par de vértices en un árbol de recubrimiento de coste mínimo no es necesariamente un camino de coste mínimo. En el árbol de recubrimiento de coste mínimo para el grafo anterior el camino que une el vértice 1 con el vértice 3 cuesta 3 mientras que el camino mínimo cuesta tan solo 2.

12.21. Dijkstra

Suponiendo que las aristas están valoradas con números positivos, modificar el algoritmo de Dijkstra (ver Ejercicio 12.17) para que el resultado indique el número de caminos de coste mínimo desde un origen a cada uno de los demás vértices del grafo.

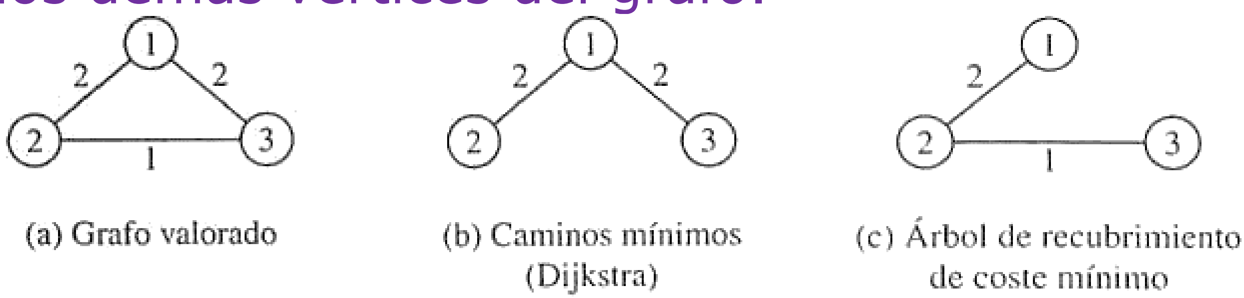


Figura 12.10: Contraejemplos para el Ejercicio 12.20.

--------------------------Solución:-----------------------

Aunque el resultado del algoritmo sea el número de caminos de coste mínimo desde el origen a cada vértice, es necesario seguir llevando el vector *coste-mín* con los costes mínimos. Las modificaciones al algoritmo consisten en inicializar y actualizar correctamente en cada etapa los valores del vector *número,* que representa el número de caminos internos desde el origen hasta cada vértice. De esta forma, el valor inicial de *número* será 1 para los vértices adyacentes al origen, y 0 para los que no lo son. Al seleccionar un nuevo vértice *v, se* estudian los nuevos caminos internos que se obtienen a partir de las aristas que salen de v. Para cada vértice *w* adyacente a v:

- Si el coste del nuevo camino es menor que el coste de los otros caminos internos, se actualiza *coste-min*[w] y ahora el número de caminos internos de coste mínimo

para *w* coincide con el número de caminos intemos de coste mínimo para *v*, ya que cada uno de estos se extiende con la arista ⟨*v*,*w*⟩.

- En el caso de que el coste del nuevo camino interno coincida justamente con el valor de *coste-min*[w], hay que incrementar *número*[w] en *número[v]* ya que ahora se tienen como caminos internos de coste mínimo todos los que había antes más todos los que hay para v, extendidos con la arista ⟨*v*, *w*⟩.
- Cuando el coste del nuevo camino interno sea mayor que el coste de los demás caminos internos para *w*, no hay que hacer nada.

{en *G* todos los vértices son accesibles desde v_1}

```
fun Dijkstra-núm(G: grafo-val[n], v1: 1..n)
dev numero[1..n] de nat
var candidatos: conjunto[1..n], coste-
mín[1..n] de valor

candidatos:= cjto-vacío();
  para i=1 hasta n hacer
    añadir(i, candidatos);
  fpara
quitar(v1, candidatos);
coste-mín[1..n]:= [+∞];
número[1..n]:= [0]
coste-mín[v1]:= 0;
número[v1]:= 1;
l:= adyacentes(v1, G);
  mientras ¬es-lista-vacía?(l) hacer
    w:= izquierdo(l);
    elim-izq(l);
```

```
    coste-mín[v]:= gv-valor(v1, v, G);
    numero[v]:= 1;
  fmientras

  para i=1 hasta n-2 hacer
    v:= mínimo(candidatos, coste-mín);
    {determinar el candidato con menor coste-mín}
    quitar(v, candidatos);
    l:= adyacentes (v, G);
    mientras ¬es-lista-vacía?(l) hacer
      w:= izquierdo(l) ;
      elim-izq(l);
      coste:= coste-min[v] + gv-valor(v, w, G);
      casos
         coste-min[w]>coste → coste-mín[w]:=
        coste; número[w]:= número[v];
        □ coste-mín[w]=coste → numero[w]:=
        número[w] + número[v];
        □ coste-mín[w]<coste → nada;
      fcasos
    fmientras
  fpara
ffun
```

El coste de este algoritmo modificado es del mismo orden que el del algoritmo original, ya que las modificaciones solo implican costes constantes adicionales. Se pueden realizar versiones como las dadas para el algoritmo de Dijkstra en los Ejercicios 12.18 y 12.19.

Capitulo 13.

13.PROGRAMACIÓN DINÁMICA

ESQUEMA GENERAL

En muchos casos, el planteamiento recursivo obtenido mediante el método de divide y vencerás conduce a la repetición de llamadas recursivas, es decir, a la resolución de los mismos subproblemas múltiples veces, lo que implica costes (en tiempo) exponenciales. A modo ilustrativo, consideremos el cálculo de los coeficientes binomiales mediante la conocida fórmula recurrente:

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \lor k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \end{cases}$$

Figura 13.1: Árbol de llamadas recursivas para calcular (6 4).

La Figura 13.1 muestra las llamadas recursivas a realizar para calcular $\binom{6}{4}$. Se

observa claramente cómo se calculan una y otra vez los mismos coeficientes. La consecuencia es que el coste en tiempo de un algoritmo que implementase directamente esta definición recursiva sería exponencial.

¿Cómo conseguir un algoritmo mucho más eficiente utilizando la misma recurrencia? La solución es utilizar una tabla para ir almacenando los coeficientes binomiales que se van calculando para, de esta forma, calcularlos una única vez. Esto nos lleva a pasar del diseño *recursivo descendente* a un diseño *iterativo ascendente*, donde se comienza por resolver todos los subproblemas más pequeños que se puedan necesitar, para ir combinándolos hasta llegar a resolver el problema original. En el ejemplo de los coeficientes binomiales, la tabla corresponde al famoso ***triángulo*** *de* ***Pascal****.*

$$\binom{0}{0}$$
$$\binom{1}{0} \quad \binom{1}{1}$$
$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$
$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$
$$\vdots$$

donde para obtener un elemento solo es necesario sumar los 2 elementos "superiores", y el coste en tiempo para calcular $\binom{n}{k}$ está en Θ(nk) (véase Ejercicio 13.18).

Por tanto, la base de la programación dinámica es el uso de una tabla para ir almacenando los resultados correspondientes a instancias más sencillas del problema a resolver. Para la aplicación de este método,

se puede seguir el siguiente esquema, consistente en 2 etapas diferenciadas:

Identificación

- Especificación de la función que representa el problema a resolver.
- Determinación de las ecuaciones recurrentes para calcular dicha función.
- Comprobación del alto coste de cálculo de dicha función debido a la repetición de subproblemas a resolver.

Construcción

- Sustitución de la función por una tabla.
- Inicialización de la tabla según los casos base de la definición recursiva de la función.
- Sustitución, en las ecuaciones, de las llamadas recursivas por consultas a la tabla.
- Planificación del orden de llenado de la tabla, de forma que se respeten las necesidades de cada entrada de la tabla.

En el caso de problemas de ***optimización***, la aplicación de esta técnica exige que se cumpla el *principio de optimalidad de Bellman* según el cual para conseguir una solución óptima basta considera subsoluciones óptimas.

En numerosas ocasiones, la función/tabla que representa el problema a resolver corresponde solamente al valor asociado a la solución que se desea encontrar, pero dicha solución no se almacena de forma explícita. Sin embargo, la tabla calculada sí contiene implícitamente la información necesaria para obtener dicha solución, por lo que basta diseñar un algoritmo adicional que, a partir de la tabla, construya la correspondiente solución (véase, por ejemplo, el Ejercicio

13.1).

En la mayoría de los casos, la utilización de la tabla reduce drásticamente el coste en tiempo de los algoritmos, si bien se incrementa de forma importante el coste en espacio adicional. En ocasiones es posible mejorar el coste en espacio, de forma que, en cada momento, en lugar de mantener la tabla completa, solo se va guardando la parte estrictamente necesaria (véase, por ejemplo, el Ejercicio 13.1).

EJERCICIOS RESUELTOS PD

13.1. Monedas

Sea un conjunto finito $M = \{m_1,\ldots,m_n\}$ de tipos de monedas, donde cada mi es un número natural y queremos pagar una cantidad **C** > 0 utilizando el menor número posible de monedas. Escribir un algoritmo que resuelva el problema. (Suponer que existe una cantidad ilimitada de monedas de cada valor).

-----------------Solución-------------------

(En el Ejercicio 12.6 vimos una estrategia voraz para resolver este problema, pero no siempre daba lugar, en general, a una solución óptima).

Para resolver el problema mediante programación dinámica, definimos una función que considere las distintas posibilidades y se quede con la mejor:

monedas(n, C) = número *mínimo* de monedas para pagar la cantidad *C* considerando los tipos de monedas del 1 al *n*.

Para calcular la función anterior consideramos el tipo de moneda *n*, con valor m_n. Existen 2 posibilidades:

1. El valor de esta moneda supera la cantidad *C* a pagar, en cuyo caso tendremos que pagar la cantidad *C* con el resto de las monedas.
2. El valor m_n es $\leq C$ en cuyo caso podemos plantear 2 opciones:
   - 2.1. No cogemos ninguna moneda de tipo *n* y seguimos como en el caso anterior, o
   - 2.2. Cogemos una moneda de tipo *n* y pagamos el resto, $C\text{-}m_n$,

considerando de nuevo las monedas de los tipos 1 a *n.*

Como estamos tratando de minimizar el número de monedas, nos quedamos con la opción que elija menos monedas.

Estas posibilidades quedan reflejadas en la siguiente fórmula:

$$monedas(n, C) = \begin{cases} solución(n-1, C) & \text{si } m_n > C \\ min\{solución(n-1, C), solución(n, C-m_n)+1\} & \text{si } m_n \leq C \end{cases}$$

donde en la parte derecha hemos escrito *solución* para referirnos a una forma cualquiera de resolver el problema con los parámetros apropiados en cada caso. Ahora bien, en la primera posibilidad (m_n>*C*) es obvio que la solución, al coincidir con la de la izquierda, tiene que ser óptima, por lo que podemos utilizar recursivamente la misma función *monedas* que estamos definiendo. En la segunda posibilidad *(m_n ≤ C)* el valor óptimo buscado va a coincidir con una solución o la otra, por lo que al menos una de las dos tiene que ser óptima, pero en principio no sabemos cuál. Sin embargo, si una de ellas no es óptima se puede mejorar y esto podría mejorar el valor total: por ejemplo, si *monedas(n*-1, C) ≤ *solución(n*-1,C), entonces

$$\text{mín}\{monedas(n-1, C),\ monedas\ (n, C-m_n)+1\} \leq min\{solución(n-1, C),\ monedas\ (n, C-m_n)+1\}.$$

Por tanto, no perdemos ningún valor óptimo si consideramos solamente soluciones óptimas para los subproblemas (es decir, se cumple el principio de optimalidad) y consideramos una definición recursiva de la

siguiente forma:

$$monedas(n, C) = \begin{cases} \text{monedas}(n-1, C) & \text{si } m_n > C \\ \text{min}\ \{\text{monedas}(n-1, C),\ monedas\ (n, C - m_n) + 1\} & \text{si } m_n \leq C \end{cases}$$

Esta fórmula define recursivamente el valor de la función *monedas* para valores concretos *n* y C. El mismo razonamiento sirve para justificar su generalización cuando estamos considerando los tipos de monedas del 1 al *i y* nos queda por pagar una cantidad j:

$$monedas(i, j) = \begin{cases} monedas\ (i-1, j) & \text{si } m_i > j \\ \text{mín}\ \{monedas\ (i-1, j),\ monedas\ (i, j - m_i) + 1\} & \text{si } m_i \leq j \end{cases}$$

donde $1 \leq i \leq n$ y $1 \leq j \leq C$. Notemos que la recursión está bien fundada porque uno u otro argumento decrece estrictamente.

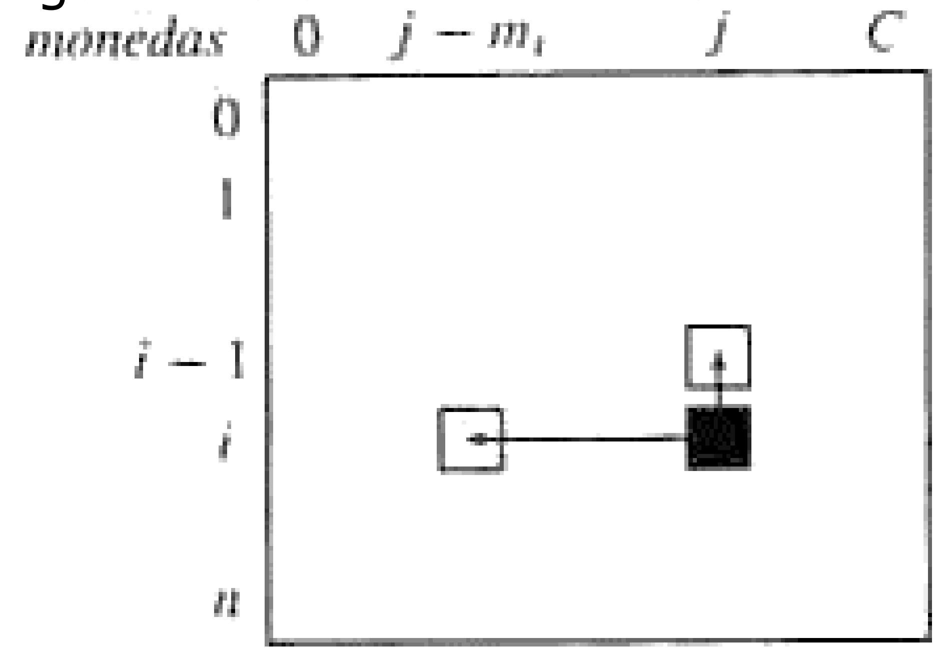


Figura 13.2: Esquema de tabla para el problema de las monedas.

Los casos básicos de la recurrencia se presentan cuando no tenemos ya más tipos de monedas que considerar pero aún tenemos que pagar cierta cantidad mayor que 0. o bien cuando no nos falta nada por pagar. En el primer caso no existe solución, lo que vamos a representar con el valor +∞ porque el problema es de minimización y así cualquier otra opción será mejor (+∞ es el elemento neutro del mínimo, extendiendo las operaciones aritméticas apropiadamente). En

el segundo caso, el número de monedas necesario será 0. Partimos, por tanto, de

$monedas(0, j) = +\infty \qquad 1 \le j \le C$

$monedas(i, 0) = 0 \qquad 0 \le i \le n.$

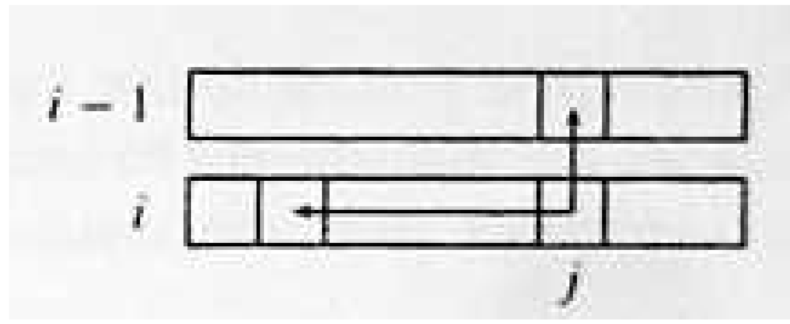
Calculamos de forma ascendente los valores de la recurrencia, desde los casos básicos a los casos mayores con ayuda de una tabla *monedas*[0..*n*, 0..C] donde vamos guardando los resultados calculados. Para ver cómo recorremos la tabla para rellenarla, la Figura 13.2 muestra qué posiciones necesitamos tener calculadas previamente cuando vamos a calcular la posición *(i, j)*. Vemos que para calcular la posición *(i, j)* necesitamos la posición de la misma columna *j* en una fila anterior, i-1, y una posición en la misma fila *i* y en una de las columnas anteriores a *j*. Podemos entonces recorrer la matriz por filas de arriba abajo y cada fila de izquierda a derecha. La solución, el menor número de monedas con las que podemos pagar la cantidad C. será el valor de la posición *monedas*[*n*, C] de la tabla, una vez que esta esté completamente rellenada. El algoritmo que rellena la tabla es el siguiente:

```
fun devolverCambio1(M[1..n] de nat⁺, C: nat) dev número: nat∞
{número es la cantidad de monedas en la solución óptima}
{número es +∞ cuando no existe solución}
var monedas[0..n, 0..C] de nat∞

{inicialización}
monedas[0, 1..C] := [+∞];
monedas[0..n, 0] := [0];
  {rellenar la matriz}
  para i = 1 hasta n hacer
    para j = 1 hasta C hacer
      si M[i] > j entonces
```

```
      monedas[i. j]:= monedas[i-1,j];
    si no
      monedas[i, j]:= mín(monedas[i-1, j],
      monedas[i, j–M[i]]+1);
    fsi
  fpara
fpara
número:= monedas[n, C];
ffun
```

El coste del algoritmo está en Θ(nC) tanto en tiempo de ejecución como en espacio adicional. Sin embargo, podemos optimizar el coste en espacio adicional utilizando solo un vector, *monedas[0..C],* porque cuando estamos rellenando la fila i, para cada posición solo necesitamos un valor de la fila anterior *i* - 1 (en la misma columna) y un valor de la misma fila. Es decir que las filas anteriores a la *i* - 1 no son necesarias.



Si ahora suponemos que el vector tiene la fila *i*-1 de la matriz, y estamos rellenando la fila *i* de izquierda a derecha, podemos hacer la actualización sobre el mismo vector ya que al ir a rellenar la posición *j* tendremos en esa posición el valor de la fila anterior, porque todavía no la hemos actualizado, y en la posición necesaria más a la izquierda el valor de la fila í, porque ya ha sido actualizada. Además, el valor de la columna *j* y fila *i*-1 no vuelve a ser utilizado, por lo que el valor calculado *monedas(i, j)* puede ser guardado en la posición *j* del vector.

El algoritmo utilizando un solo vector

es el siguiente:

```
fun devolverCambio2(M[1…n] de nat+, C: nat) dev número : nat∞
var monedas[0…C] de nat∞.

{inicialización}
monedas[0]:= 0;
monedas[1..C]:= [+∞];
  {hacer las actualizaciones}
  para i = 1 hasta n hacer
    para j = M[i] hasta C hacer
      monedas[j]:= mín(monedas[j],monedas[j-M[i]] + 1);
    fpara
  fpara
número:= monedas[C];
ffun
```

Obsérvese que el bucle más interno empieza con *j* = M[i] ya que para los valores menores no hay que hacer ninguna modificación sobre el vector (en la matriz, la fila *i*-1 y la *i* coincidían para estos valores de y). El coste en tiempo de esta nueva versión sigue estando en Θ(nC) pero el coste en espacio adicional se ha reducido a Θ(C).

Supongamos ahora que. además del número total de monedas en la solución óptima, queremos conocer también dicha solución, o sea, CUÁNTAS monedas de cada tipo la forman. Veamos que una vez rellenada la tabla podemos recuperar dicha información. Cuando actualizábamos la posición *monedas[i. j* y cabía la posibilidad de tomar una moneda de tipo *i.* hacíamos

$$\text{monedas}(i,j) = \text{mín}\{\underbrace{\text{monedas}(i-1,j)}_{\text{no cogemos moneda } m_i}, \underbrace{\text{monedas}\ (i,j-m_i)+1}_{\text{si cogemos moneda } m_i}\}$$

Por tanto, sabemos que si

$monedas(i, j) = monedas(i–1, j)$

es porque podemos no coger ninguna moneda de tipo *i* para pagar la cantidad j, mientras que si

$monedas(i, j) \neq monedas(i–1, j)$

debemos coger al menos una moneda de tipo *i* para pagar la cantidad y. Pero haciendo esta comparación necesitamos la fila anterior, *i* – 1, que no tenemos si hacemos la optimización de espacio adicional. Sin embargo, podemos hacer la comparación contraria: si tenemos

$monedas(i, j) = monedas(i, j\text{-}m_i) + 1$

es porque podemos coger al menos una moneda de tipo *i* para pagar la cantidad y. mientras que si

$monedas(i,j) \neq monedas(i, j – m_i) + 1$

es porque no debemos coger ninguna moneda de tipo *i* para pagar la cantidad *j.* Y con esta comparación solo necesitamos la información que mantenemos en una fila. En ella podemos ir haciendo comparaciones desde la posición *C* hacia la izquierda, para saber cuántas monedas de cada tipo hay que coger.

De hecho, este razonamiento no es suficiente para justificar el siguiente algoritmo, puesto que al tener solo el vector *monedas[0..C]* no tenemos información de cada fila, sino solamente de la última. Sin embargo, la idea sigue siendo válida porque la última fila contiene la información sobre el número de monedas mínimo para cada cantidad, con el sistema monetario completo. Si $monedas(n, j) = monedas(n, j\text{-}m_i) + 1$ para algún *j y* algún *i,* sabemos que podemos coger una moneda de tipo *i* para conseguir una solución óptima para pagar *j.* Además, como el número de monedas de cada tipo es

ilimitado, el sistema monetario *no cambia* y se puede iterar el proceso para *j* - m_i, haciendo comparaciones de nuevo en la última fila, es decir, el vector. Para implementar este proceso, al principio *j* vale *C* e *i* vale n; la cantidad *j* decrece tal y como se van cogiendo monedas y, cuando la ecuación anterior no es cierta y por tanto no se puede coger ninguna moneda más de tipo *i. se* decrementa *i* pasando a considerar el tipo de moneda anterior *i* - 1.

La función que calcula siguiendo este método cuántas monedas de cada tipo son necesarias a partir del vector *monedas* calculado anteriormente, y que solo tiene sentido cuando monedas[C] ≠ +∞. es decir, cuando existe solución, es la siguiente:

```
{monedas[C] ≠ +∞ }
fun calcularMonedas(M[1…n] de nat+,
monedas[0..C] de nat∞) dev cuántas[1 ..n] de
nat

{cuántas[i] es el número de monedas de tipo i cogidas}
cuántas [1…n]:= [0]
i:= n;
j:= C;
  mientras j>0 hacer {no hemos pagado todo}
    si M[i] ≤ j ∧c monedas[j] = monedas[j - M{i]]
    +1 entonces
      {tomamos una moneda de tipo i }
      cuántas[i]:= cuántas[i] + 1;
      j:= j - M[i];
    si no {no tomamos más monedas de tipo i}
      i:= i – 1;
    fsi
  fmientras
ffun
```

El coste de esta función está en Θ(C) en tiempo y en Θ(1) en espacio adicional.

13.2. Mochila no fraccionable

Cuando Alí-Babá consigue por fin entrar en la Cueva de los 40 Ladrones encuentra allí gran cantidad de objetos muy valiosos. A pesar de su pobreza, Alí-Babá conoce muy bien el peso y valor de cada uno de los objetos en la cueva. Debido a los peligros que tiene que afrontar en su camino de vuelta, solo puede llevar consigo las riquezas que quepan en su pequeña **MOCHILA**, que soporta un peso máximo conocido.

(a) Suponiendo que **los objetos no se pueden fraccionar** y que los pesos son números naturales positivos, determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila. Los valores son reales positivos.

(b) Supongamos ahora que en la cueva hay una cantidad inagotable de riquezas de diferentes clases. Cada riqueza de una clase es **indivisible** y tiene cierto *peso* (natural positivo) y cierto *valor* (real positivo) conocidos. Determinar qué cantidad de riquezas de cada clase debe coger Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

(c) Generalizar el primer apartado suponiendo pesos reales.

---------------------------Solución-----------------------

Apartado (a)

Supongamos que en la cueva hay *n* objetos, cada uno con un peso (natural) $p_i > 0$ y un valor (real) $v_i > 0$ para todo *i* entre 1 y *n.* y que la mochila soporta un peso total (natural) máximo $M > 0$.

El problema consiste en maximizar $\sum_{i=1}^{n} x_i v_i$ con la restricción $\sum_{i=1}^{n} x_i p_i \leq M$ donde xi ∈ {0, 1} dice si hemos cogido (1) o no (0) el objeto *i.*

En el Ejercicio 12.5 vimos una solución voraz a este problema cuando los objetos se pueden partir ($0 \leq x_i \leq 1$) mientras que ahora los objetos son indivisibles.

Para resolver el problema definimos una función

mochila(i, j) = máximo valor que podemos poner en la mochila de peso máximo *j* considerando los objetos del 1 al *i.*

Calcularemos esta función mediante una recurrencia que va a tener en cuenta las distintas posibilidades. Consideramos primero el objeto *i.* Tenemos 2 posibilidades:

1) Que no podamos coger el objeto *i* porque su peso supera el peso que todavía podemos añadir a la mochila, en cuyo caso tendremos que llenar la mochila con los objetos restantes, del 1 al *i* - 1.
2) Que sí podamos cogerlo, en cuyo caso podemos planteamos 2 opciones:
   2.1. No lo incluimos, y seguimos probando con el resto de los

objetos, o

2.2. Sí lo cogemos, obteniendo un valor vi, y llenamos el resto de la mochila (j-p_i) con el resto de los objetos.

Como estamos tratando de maximizar el valor de la mochila, nos quedamos con la posibilidad de mayor valor.

Una justificación completamente análoga a la vista en la solución del Ejercicio 13.1 para la función *monedas* permite definir de forma recursiva la función *mochila* al cumplirse el principio de optimalidad, de forma que basta considerar solamente soluciones óptimas para los subproblemas. Así pues, tenemos la siguiente recurrencia:

$$\text{mochila}(i,j) = \begin{cases} \text{mochila}(i-1,j) & \text{si } p_i > j \\ \text{máx}\{\text{mochila}(i-1,j), \text{mochila}(i-1,j-p_i)+v_i\} & \text{si } p_i \leq j \end{cases}$$

donde $1 \leq i \leq n$ y $1 \leq j \leq M$, y la reclusión está bien fundada porque uno o los 2 argumentos decrecen estrictamente.

Los casos básicos se presentan, bien cuando no tenemos objetos que considerar, o bien cuando no nos queda espacio en la mochila. En ambos casos el único beneficio posible es 0:

$$mochila(0, j) = 0 \quad 0 \leq j \leq M$$
$$mochila(i, 0) = 0 \quad 0 \leq i \leq n.$$

Calcularemos los valores *mochila(i, j)* con ayuda de una tabla *mochila[0..n,* 0..*M*]. La Figura 13.3 muestra qué posiciones necesitamos tener calculadas cuando vamos a rellenar la posición (i, j). lo que nos sirve para saber cómo

debemos recorrer la tabla para rellenarla. Para calcular la posición *mochila[i, j]* necesitamos haber calculado dos posiciones de la fila anterior (*i*-1). Podemos recorrer la matriz por filas de arriba abajo y cada fila de izquierda a derecha (o de derecha a izquierda). Al terminar de rellenar la matriz, *mochila[n, M]* contendrá el beneficio de la solución óptima.

Si solo quisiéramos el valor máximo alcanzable, podríamos optimizar el espacio adicional utilizando solo un vector (como hicimos en el Ejercicio 13.1) que recorreríamos en este caso *de derecha a izquierda.* para no perder los valores de la fila anterior que todavía se necesitan (véase la solución del Ejercicio 13.4).

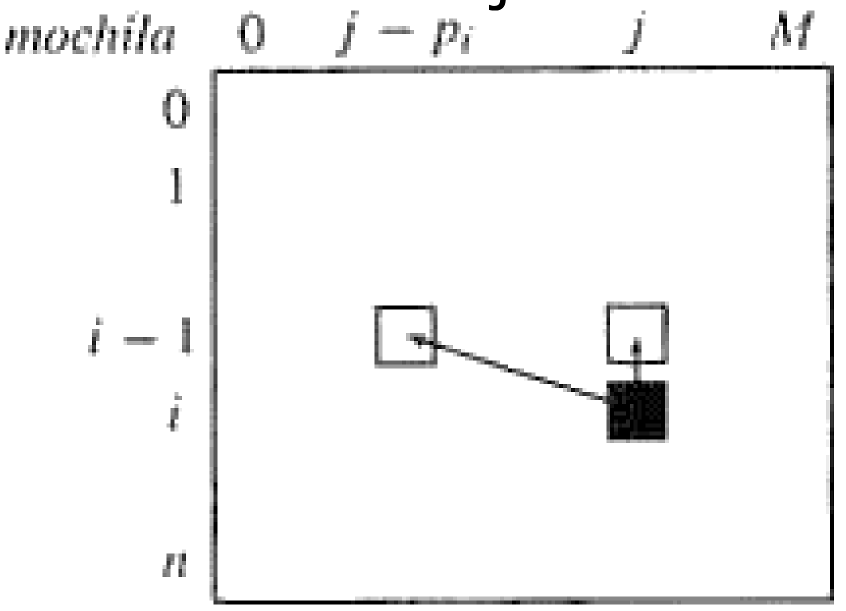


Figura 13.3: Esquema de tabla para el problema de la mochila.

En cambio, si queremos también devolver los objetos que forman parte de la solución óptima no interesa optimizar, porque en ese caso las comparaciones que hacemos para llenar una posición siempre se refieren a posiciones de la fila anterior:

$$\text{mochila}(i,j) = \text{máx}\{\underbrace{\text{mochila}(i-1,j)}_{\substack{\text{no cogemos}\\ \text{el objeto } i}}, \underbrace{\text{mochila}(i-1,j-p_i)+v_i}_{\substack{\text{sí cogemos}\\ \text{el objeto } i}}\}$$

Hay que decir que aun optimizando podríamos recuperar la solución si guardásemos las decisiones tomadas en otra matriz de decisiones. Sin embargo, de esta forma no mejoraríamos el coste en espacio adicional del algoritmo.

El algoritmo que calcula la solución implementando estas ideas es:

```
fun mochilaPd(P[1..n] de nat+, V[1...n] de
real+, M : nat+) dev ⟨valor : real,
cuáles[1..n] de 0..1⟩
{cuáles[i] indica si hemos cogido o no el objeto i}
var mochila[0..n, 0..M] de real

{inicialización }
mochila[0..n, 0] := [0];
mochila[0, 1..M] := [0];

{ rellenar la matriz }
para i = 1 hasta n hacer
   para j = 1 hasta M hacer
      si P[i] > j entonces
         mochila[i, j] := mochila[i-1, j];
      si no
         mochila[i, j] := máx(mochila[i-1, j],
         mochila[i-1, j-P[i] + V[i]);
      fsi
   fpara
fpara
valor:= mochila[n, M];

{cálculo de los objetos}
resto:= M;
para i = n hasta 1 paso -1 hacer
   si mochila[i, resto] = mochila[i-1, resto]
   entonces
```

```
      {no cogemos el objeto i}
      cuáles[i]:= 0;
    si no {sí cogemos el objeto i}
      cuáles[i]:= 1;
      resto:= resto - P[i];
    fsi
  fpara
ffun
```

El coste de la función mochila-pd, tanto en tiempo como en espacio adicional, está en Θ(nM).

Obsérvese que en el caso de que los pesos fueran números reales en vez de naturales no se podrá utilizar esta solución ya que los pesos resultantes no podrían utilizarse para indexar las columnas de una matriz. Presentaremos una solución a este caso en el Apartado (c).

Apartado (b)

Supongamos que en la cueva hay objetos de *n* clases diferentes. Este apartado es similar en principio al anterior, solo que en este caso tenemos clases en vez de objetos y una cantidad inagotable de objetos de cada clase, por lo que, como veremos, se convierte en un problema análogo al de las monedas (Ejercicio 13.1). Definimos la siguiente función:

Riquezas(i, j) = *máximo* valor que podemos conseguir con una mochila de peso máximo *j* considerando las *i* primeras clases de objetos.

Para definir la recurrencia correspondiente, consideramos la clase de objetos *i.* Tenemos dos posibilidades, o no coger ningún objeto de esta clase, y entonces llenar la mochila de manera óptima con los objetos de las *i* - 1 clases restantes, o coger un objeto de la clase *i* (si cabe), obteniendo un valor v_i, y llenar el resto de la mochila (j - ti) de manera óptima con objetos de cualquiera de las clases hasta la *i* inclusive. Los subproblemas deben resolverse también de forma óptima puesto que se cumple el principio de optimalidad como en el apartado anterior. Por tanto, tenemos que:

$$\text{riquezas}(i,j) = \begin{cases} \text{riquezas}(i-1,j) & \text{si } p_i > j \\ \text{máx}\{\text{riquezas}(i-1,j), \text{riquezas}(i,j-p_i)+v_i\} & \text{si } p_i \le j \end{cases}$$

donde $1 \le i \le n$ y $1 \le j \le M$. Nótese que la única diferencia entre la recurrencia para la función *riquezas* y la función

mochila del apartado anterior es que en el caso de coger un objeto allí teníamos *i*-1 como primer argumento mientras que ahora tenemos *i;* esto corresponde a la idea de que se puede continuar cogiendo objetos de la misma clase.

Los casos básicos se presentan cuando no tenemos ya más clases de objetos, o cuando no podemos añadir más peso a la mochila. En ambos casos el beneficio es 0:

$$\begin{array}{lll} \text{riquezas}(0, j) & = 0 & 0 \leq j \leq M \\ \text{riquezas}(i, 0) & = 0 & 0 \leq i \leq n. \end{array}$$

El valor máximo que puede obtener Alí-Babá se obtiene con la llamada *riquezas(n, M).*

Calculamos estos valores con ayuda de una tabla, *riquezas[0..n*, 0..M]. La forma en la que hay que recorrer la tabla es la misma que para el problema de las monedas, es decir, por filas de arriba abajo y, cada fila, de izquierda a derecha. Además, incluso teniendo que calcular los objetos que forman parte de la mejor solución, podemos optimizar el espacio adicional necesario utilizando solo un vector (como hemos visto en la solución del Ejercicio 13.1). por lo que logramos un algoritmo de coste en $\Theta(nM)$ en tiempo y en $\Theta(M)$ en espacio adicional. El algoritmo que implementa estas ideas es el siguiente:

```
fun mochilaClasesPd(P[1..n] de nat⁺, V[1..n]
de real⁺, M : nat⁺) dev ⟨valor: real, cuántos[1..n]
de nat⟩
```

```
{cuántos[i] es el número de objetos tomados de clase i}
var riquezas[0..M] de real

{inicialización}
riquezas[0..M]:= [0];
  {actualizaciones del vector}
  para i = 1 hasta n hacer
    para j = P[i] hasta M hacer
      riquezas[j]:= máx(riquezas[j], riquezas[j-
      P[i]]+ V[i]);
    fpara
  fpara
valor:= riquezas[M]; {riqueza máxima que puede
llevar Alí-Babá en su mochila}
{cálculo de los objetos}
cuántos[1..n]:= [0];
i:= n  {empezamos por la última clase}
resto:= M; {y tenemos la mochila vacía }
  mientras resto>0 ∧ i>0 hacer
    si resto ≥ P[i] ∧c riquezas[resto] =
    riquezas[resto - P[i]] + V[i] entonces
      {tomamos un objeto de clase i}
      cuantos[i]:= cuantos[i] + 1;
      resto:= resto - P[i];
    si no   {no tomamos más objetos de clase i}
      i:= i-1;
    fsi
  fmientras
ffun
```

Apartado (c)

La definición de la función recursiva presentada en el Apartado (a) no se ve afectada por el hecho de que los pesos sean números reales en vez de naturales (aunque en este caso, para facilitar la presentación posterior, abreviamos el nombre):

$$m(i,j) = \begin{cases} m(i-1,j) & \text{si } p_i > j \\ \text{máx}\{m(i-1,j), m(i-1,j-p_i)+v_i\} & \text{si } p_i \le j \end{cases}$$

El caso básico se da cuando no hay objetos para elegir, en cuyo caso el beneficio es nulo: $m(0, j)=0$. El valor óptimo que interesa calcular viene dado por $m(n, M)$.

Sin embargo, si M o los pesos pi no son números naturales (o sea, solo sabemos que son números reales), no se puede utilizar una matriz para calcular $m(i,j)$ puesto que tenemos una ecuación para todo peso real j con $0 \le j \le M$. Si ponemos $m(i, j) = -\infty$ para $j<0$, la fórmula anterior se simplifica a

$$m(i,j) = \text{máx}\{m(i-1,j), m(i-1,j-p_i)+v_i\}$$

donde $1 \le i \le n$ (natural) y $j \ge 0$ (real). Fijando i tenemos una serie de funciones reales m_i tales que en el caso básico

$$m_0(j) = \begin{cases} -\infty & \text{si } j<0 \\ 0 & \text{si } j \ge 0 \end{cases}$$

y en el caso recursivo, para todo j real

$$m_i(j) = \text{máx}\{m_{i-1}(j), m_{i-1}(j-p_i)+v_i\}.$$

Para una función real f y un par de reales (h,v), definimos la ***traslación*** *de f vía (h, v)* como la función real f' tal que

$$f'(j) = f(j-h) + v$$

es decir, la gráfica de f es trasladada h unidades horizontalmente a la derecha y u unidades verticalmente hacia arriba. Entonces

$$m_i = \text{máx}\{m_{i-1}, m'_{i-1}\}$$

donde m'_i denota la traslación de m_i vía (p_i, v_i).

En estas condiciones será posible utilizar el método de programación dinámica e implementar la recursión

anterior de forma iterativa, si conseguimos implementar adecuadamente el cálculo de una función real a partir de otras. El valor que interesa es $m_n(M)$ *y,* como en la implementación matricial para pesos naturales del Apartado (a), es sencillo calcular cuáles son los objetos escogidos en la solución óptima a partir de las funciones m_i.

La cuestión pendiente es: ¿Cómo representamos las funciones reales? Afortunadamente, las funciones m_i *y* m'_i tienen una forma especial: son *escalonadas crecientes positivas* (véase la Figura 5.4 en la página 143) y por tanto vienen determinadas por el conjunto de pares (r_l, s_l) que constituyen los *saltos* de la función escalonada y donde podemos suponer sin pérdida de generalidad que el primer salto es (0,0). Además el número de saltos de las funciones escalonadas m_i y m'_i es finito.

Si el conjunto de saltos S representa una función escalonada *f,* el conjunto de saltos *S'* correspondiente a la traslación de *f* vía (*w*,z) (donde *w y* z son ambos reales no negativos) se obtiene sumando el par (w,z) a todos los pares en S:

$$S' = \{(r_l + w, s_l + z) \mid (r_l, s_l) \in S\} \cup \{(0,0)\}$$

Por otra parte, la función escalonada resultante de hacer el máximo de dos funciones escalonadas es también fácil de calcular comparando los correspondientes saltos (véase el Ejercicio 5.14).

Como m_0 es una función escalonada creciente con único salto (0,0) y m_i se obtiene a partir de m_{i-1} mediante las operaciones de traslación y máximo, esto demuestra que todas las funciones mi y m'_i son efectivamente escalonadas crecientes con primer salto (0,0) y un número finito de saltos.

En el Ejercicio 5.14 se han implementado las funciones escalonadas con estas condiciones mediante listas no vacías de pares de números reales (los saltos correspondientes, incluyendo al menos el par (0,0)), *ordenadas* de menor a mayor según la coordenada x (y por tanto también ordenadas de menor a mayor según la coordenada y). Las operaciones sobre tales funciones incluyen crear la función constante nula (el único salto es el (0,0)), hacer una traslación, calcular el máximo de dos funciones y calcular el valor de una función escalonada en un punto dado. Con todo esto tenemos los ingredientes necesarios para implementar de forma iterativa la solución al problema de la mochila real mediante un vector de funciones escalonadas.

```
fun mochilaReal(P[1..n], V[1 ..n] de real+, M : real+) dev ⟨valor: real, cuáles[1..n] de 0..1⟩
var m[0..n] de funciónEscalonada,
      f: funciónEscalonada

m[0] := constante-nula();
```

```
para i = 1 hasta n hacer
  f := trasladar(m[i-1], P[i], V[i]);
  m[i] := máxFunción(m[i-1], f);
fpara

valor := aplicar(m[n],M);
resto := M;
para i = n hasta 1 paso -1 hacer
  si aplicar(m[i],resto) = aplicar(m[i-1],
  resto) entonces
    cuáles[i]:= 0;
  sino
    cuáles[i]:= 1;
    resto:= resto - P[i];
  fsi
fpara
ffun
```

El COSTE de las operaciones trasladar, máxFunción y aplicar es lineal con respecto a la longitud de la lista que representa la función, es decir, con respecto al número de saltos de la función. La operación de traslación incrementa el número de saltos en uno, mientras que en el caso peor el número de saltos del máximo de dos funciones es la suma de los números de saltos de esas dos funciones. Por tanto el espacio total ocupado por el vector *m*[0..*n*] en el caso peor es proporcional a $\sum_{i=0}^{n} 2^i = 2^{n+1} -$, y el coste en tiempo en el caso peor está también en $O(2^{n+1})$.

13.3. Franquear sellos

El país de Fanfanisflán emite *n* sellos diferentes de valores naturales positivos $s_1,\ldots,s_n$. Se quiere enviar una carta y se sabe que la correspondiente tarifa postal es *T.* ¿De cuántas formas diferentes se puede franquear exactamente la carta, si el orden de los sellos **no importa**?

-----------------------------**Solución**---------------------------

En este ejercicio tenemos que calcular el número total de formas que hay de pagar la cantidad *T* con los sellos de *n* tipos (*consideramos una <u>cantidad ilimitada</u> de sellos de cada valor*). Definimos la función

formas(n, T) = número de formas de franquear *T* con *n* tipos de sellos.

Para plantear la correspondiente recurrencia, consideremos los sellos de valor s_n. Podemos no pone ningún sello de este valor y calcular el número total de formas de franquear *T* con el resto de los sellos, o poner un sello de clase *n* y franquear el resto con cualquier tipo de sellos. Los sellos de tipo *n* solo los podemos considerar cuando $s_n \leq T$. El número total de formas será la suma de las formas de ambas posibilidades, puesto que estas son disjuntas y no hay más. Notemos que cada forma de franquear $T - s_n$ se extiende a una única forma de franquear *T* añadiendo el sello de valor s_n.

Planteamos la recurrencia en el caso general, cuando consideramos los sellos del 1 al *i* y queremos franquear una

cantidad *j:*

$$\text{formas}(i,j) = \begin{cases} \text{formas}(i-1,j) & \text{si } s_i > j \\ \text{formas}(i-1,j) + \text{formas}(i,j-s_i) & \text{si } s_i \le j \end{cases}$$

donde $1 \le i \le n$ y $1 \le j \le T$.

Los casos básicos se presentan cuando ya no nos quedan tipos de sellos que considerar ($i=0$) y todavía nos queda alguna cantidad por franquear ($j>0$), en cuyo caso no hay ninguna forma de franquear la cantidad.

$$\text{formas}(0,j) = 0 \qquad 1 \le j \le T.$$

o bien cuando la cantidad a franquear es 0, en cuyo caso la única posibilidad es no poner ningún sello:

$$\text{formas}(i,0) = 1 \qquad 0 \le i \le n.$$

Utilizamos una matriz *formas[0..n, 0..T]* para calcular los valores de la recurrencia. Para rellenar la posición (i, j) necesitamos la posición (i-1, j) en la fila anterior y misma columna y alguna posicióі en la fila *i y* en una columna a la izquierda de *j* (como en la tabla de la Figura 13.2). Podríamos rellena la matriz por filas de arriba abajo y de izquierda a derecha; pero ya que solo nos interesa el número de formas, podemos optimizar y utilizar solo un vector *formas[0..T],* que rellenamos de izquierda a derecha

El algoritmo, de coste en $\Theta(nT)$ en tiempo y en $\Theta(T)$ en espacio adicional, es el siguiente:

```
fun sellos(S[1..n] de nat+, T: nat+) dev
numFormas: nat
var formas[0..T] de nat

{inicialización}
formas[0]:= 1;
formas[1..T]:= [0];
 {actualizaciones del vector}
 para i=1 hasta n hacer
   para j=S[i] hasta T hacer
     formas[j] := formas[j] +formas[j-S[i]];
   fpara
 fpara
númFormas := formas[T];
ffun
```

13.4. Dividir botin en 2 partes

El Maqui y el Popeye acaban de desvalijar la reserva nacional de oro. Los lingotes están empaquetados en n cajas de diferentes pesos naturales positivos *pi* para *i* entre 1 y n. Como no tienen tiempo de desempaquetarlos para dividir el botín, deciden utilizar los pesos de cada una de las cajas para distribuir el botín a medias. Al cabo de un buen rato todavía no han conseguido repartirse el botín, por lo cual acuden al Teclas para saber si el botín se puede dividir en 2 partes iguales sin desempaquetar las cajas con los lingotes.

-------------------------**Solución**-------------------------

Considerando $P = \sum_{i=1}^{n} p_i$ el problema es equivalente a comprobar si es posible, sumando algunos de los pesos, obtener *P/2.*

Si *P* es impar, claramente la respuesta es que no se puede, por lo que en el resto de la solución suponemos que *P* es par. En este caso, definimos una función

sePuede(i, j) = booleano que indica si es posible sumar la cantidad *j* eligiendo algunas de las *i* primeras cajas.

Podemos plantear la siguiente recurrencia

$$\text{sePuede }(i,j) = \begin{cases} \text{sePuede }(i-1,j) & \text{si } p_i > j \\ \text{sePuede }(i-1,j) \vee \text{ sePuede }(i-1,j-p_i) & \text{si } p_i \le j \end{cases}$$

donde 1≤*i*≤*n y* 1≤*j*≤*P/2;* es decir, o no podemos considerar pi porque pi>*j, o* bien sí podemos considerarlo, en cuyo caso hay 2 posibilidades: o no se incluye

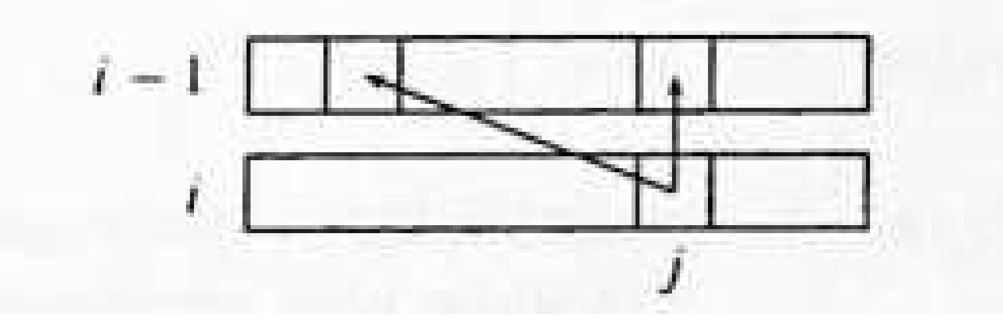
en la suma y vemos si se puede sumar *j* con el resto, o sí se incluye, por lo que pasamos a ver si se puede sumar la diferencia, j-p_i. con el resto de los pesos.

Los casos básicos se presentan cuando ya no tenemos más pesos que considerar o cuando la cantidad que queremos sumar es 0:

sePuede(0, j) = falso $1 \le j \le P/2$

sePuede(i,0) = cierto $0 \le i \le n$.

Utilizamos una matriz *sePuede[0..n, 0..P/*2] para calcular los valores de la recurrencia. Si solo queremos saber si se puede o no sumar la cantidad *P/2, y* no los objetos necesarios para sumar tal cantidad, podemos optimizar el espacio adicional utilizando solo un vector, *sePuede[0..P/2],* Ya que las 2 posiciones necesarias para calcular *sePuede(i, j)* se encuentran en la fila anterior,



el vector tiene que recorrerse *de derecha a izquierda,* para no perder valores de la fila anterior que se necesitan después.

El algoritmo cuando P es **par**, de coste en tiempo en Θ(nP) y en espacio adicional en Θ(P), es el siguiente:

```
{P' = P/2}
fun repartirBotínParPd(peso[1..n] de
nat+, P': nat+) dev respuesta: bool
var sePuede[0..P'] de bool
```

```
{inicialización}
sePuede[0]:= cierto;
sePuede[1..P']:= [falso];
{actualizaciones del vector}
para i=1 hasta n hacer
  para j=P' hasta peso[i] paso - 1 hacer
    sePuede[j] := sePuede[j] ∨ sePuede[j-
    peso[i]];
  fpara
fpara
respuesta := sePuede[P'];
ffun
```

El algoritmo principal que llama al anterior es:

$\{P = \sum_{i=1}^{n} \text{peso}[i]\}$

```
fun repartirBotínPd(peso[1..n] de nat⁺, P
: nat⁺) dev respuesta: bool

si impar(P) entonces
  respuesta := falso;
si no
  respuesta:= repartirBotínParPd(peso, P div
  2);
fsi
ffun
```

13.5. Competición

Sea una competición entre dos equipos, A y B, en la que el ganador es el primer equipo que consiga n victorias. Suponemos que no hay empates, que los resultados de todos los partidos son independientes, y que para cualquier partido dado hay una probabilidad constante p de que lo gane el equipo A (y por tanto una probabilidad constante $q = 1-p$ de que lo gane el equipo B).

(a) Desarrollar un algoritmo para calcular, antes del primer partido, la probabilidad de que el equipo A gane la competición.

(b) Hacer lo mismo cuando existe una probabilidad p de que A gane un partido, q de que lo pierda, y r (con $p+q+r=1$) de que haya empate. Un empate no supone una victoria para ningún equipo y siguen siendo necesarias n victorias para ganar la competición.

---------------------------Solución------------------------

Apartado (a)

Un estado para este problema viene determinado por el número de partidos que ha ganado cada equipo en cierto momento, o equivalentemente, por el número de victorias que le faltan a cada uno para llegar a n y entonces ganar la competición.

Definimos una función

$P(i, j)$ = probabilidad de que A gane la competición si le faltan i victorias

para ganar y a B le faltan *j* victorias

El valor que nos interesa calcular es $P(n,n)$.

La definición recursiva de la función viene dada por

$$P(i,j) = pP(i-1, j) + qP(i, j-1).$$

pues en el primer caso gana A con probabilidad *p* y le falta una victoria menos, mientras que B se queda igual; y en el segundo caso gana B con probabilidad *q y* le falta una victoria menos mientras que A se queda igual.

Los casos básicos son

$$P(0,j) = 1 \quad 1 \leq j \leq n,$$

porque si a A le faltan 0 victorias quiere decir que ya ha ganado la competición, y

$$P(i,0) = 0 \quad 1 \leq i \leq n,$$

porque si B gana la competición es imposible que la gane A. La situación $P(0, 0)$ no puede darse.

Podemos rellenar una tabla P[0..n, 0..n] por filas de 0 a *n, y* dentro de cada fila por columnas de 0 a *n.* Dejando aparte los casos básicos, para calcular cada entrada (i,*j)* en la tabla se necesitan las entradas (i-1, *j)* e (i, *j*-1), por lo que el espacio adicional se puede reducir a un vector que se rellena *de izquierda a derecha.* De esta forma se llega al siguiente algoritmo, cuyo coste en tiempo está en $\Theta(n^2)$ v en espacio adicional en $\Theta(n)$.

$\{0 \leq p \leq 1\}$

```
fun competición(n: nat+, p: real) dev
probabilidadA: real
var P[0..n] : real

P[0]:= 0;
P[1..n]:= [1];
para i=1 hasta n hacer
  para j=1 hasta n hacer
    P[j]:= p*P[j] + (1-p)*P[j-1];
  fpara
fpara
probabilidadA = P[n];
ffun
```

Apartado (b)-----------------------

Si existe la posibilidad de empatar con probabilidad r, la fórmula

$$P(i,j) = pP(i-1,j) + qP(i,j-1)$$

ya no es correcta porque no cubre el tercer caso posible; hay que modificarla como sigue:

$$P(i.j) = pP(i-1,j) + qP(I, j-1) + rP(I, j).$$

donde el tercer caso indica que al haber empatado, tanto a A como a B Ies faltan las mismas victorias que antes de jugar ese partido. Obviamente, esta fórmula no da lugar a una definición recursiva inmediata porque *P(i, j)* se llama a sí misma con los mismos argumentos. Sin embargo, teniendo en cuenta que $r<1$ (si no, los equipos siempre empatan y ninguno puede llegar a ganar nunca la competición), podemos despejar P(i,j) obteniendo

$$P(i,j) = \frac{p}{1-r} P(i-1,j) + \frac{q}{1-r} P(i,j-1).$$

que se puede resolver exactamente de la

misma forma que en el caso anterior, pues la única diferencia es el valor de los coeficientes que multiplican.

13.6. ()Multiplicar matrices

El producto de una matriz A_{pxq} y una matriz B_{qxr} es una matriz C_{pxr} cuyos elementos son

$$c_{ij} = \sum_{k=1}^{q} a_{ik} b_{kj}$$

Por tanto, se necesitan *pqr* multiplicaciones entre escalares para calcular C.

Si ahora se quiere multiplicar una secuencia de matrices $M_1 \ldots M_n$, donde cada matriz *Mi* tiene dimensiones d_{i-1} x d_i, el orden de las matrices no se puede alterar, pero sí el de los productos a realizar, ya que la multiplicación de matrices es asociativa.

Desarrollar un algoritmo que inserte paréntesis en la secuencia de matrices de forma que el número total de multiplicaciones entre escalares sea mínimo.

-------------------------Solución-------------------------

Si decidimos que el último producto entre matrices que vamos a realizar es el que esta entre la matriz *k* y la matriz *k*+1, es decir, ponemos paréntesis de la forma

$$\underbrace{(M_1 \cdots M_k)}_{d_0 \times d_k} \underbrace{(M_{k+1} \cdots M_n)}_{d_k \times d_n}$$

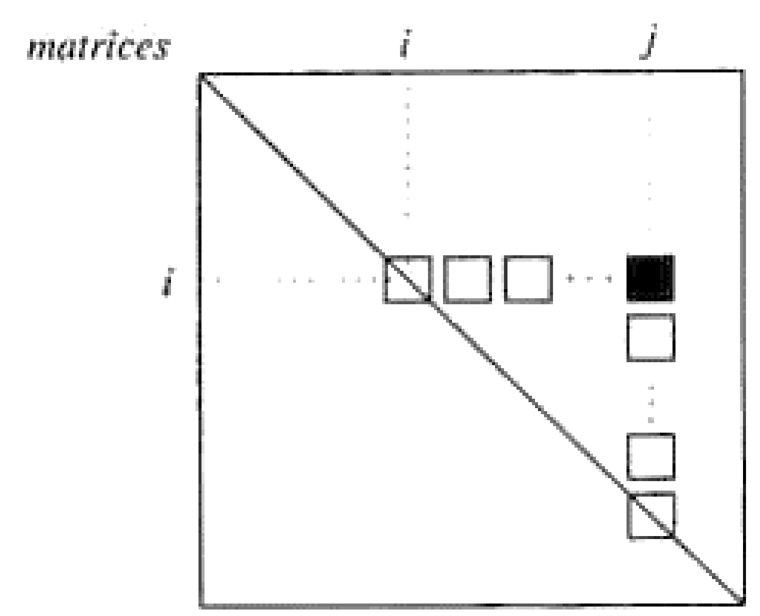


Figura 13.4: Esquema de tabla para el problema del producto matricial.

el último producto multiplicará una matriz d_0 x d_k por otra d_k x d_n, realizando $d_0d_kd_n$ multiplicaciones entre escalares. El número total de multiplicaciones escalares se obtendrá sumando a $d_0d_kd_n$ el número total de tales multiplicaciones en M_1 ... M_k y el número total en M_{k+1} ... M_n. Para que esta suma sea óptima, como la cantidad $d_0d_kd_n$ está fija, también tienen que ser óptimos los números totales correspondientes a las dos subsecuencias. Por tanto, se cumple el principio de optimalidad. Ahora bien, la posición *k* por donde partimos puede variar desde la 1 hasta la *n* - 1, por lo que necesitamos calcular el mínimo al variar *k* para obtener el valor óptimo.

Definiremos de forma recurrente la función

matrices(i, j) = número *mínimo* de multiplicaciones escalares para realizar el producto matricial *Mi . . . Mj.*

La recurrencia solo tiene sentido cuando $i \leq j$. El caso recursivo, $i<j$, se define de la siguiente manera:

$$\text{matrices}(i,j) = \min_{i \le k \le j-1} \{ \text{matrices } (i,k) + \text{matrices}(k+1,j) + d_{i-1} d_k d_j \}.$$

La recursión está bien fundada, porque tanto *k-i* como *j- (k*+1) son menores que *j-i,* es decir, en cada llamada recursiva se consideran menos matrices. El caso básico se presenta cuando solo tenemos una matriz, esto es, *i=j.* por lo que no realizamos multiplicación alguna:

*matrices(i,*i) = 0.

Utilizaremos una tabla *matrices[*1..*n*, 1..*n*] para calcular los valores *matrices(i, j)* de la cual solo necesitaremos la mitad superior por encima de la diagonal principal. La Figura 13.4 muestra para una casilla (í, *j)* cualquiera qué otras casillas necesitamos haber calculado previamente (necesarias para resolver la recurrencia). Vemos que para calcular la casilla (i,j) necesitamos los elementos de la fila *i* en columnas anteriores a la j y los elementos de la columna *j* en filas posteriores a la *i* (todo ello en la mitad que se considera). Podemos entonces rellenar la matriz recorriéndola por diagonales (entre otras posibilidades), para lo que necesitamos numerar las diagonales y los elementos dentro de cada diagonal.

La Figura 13.5 muestra una manera de numerar por diagonales, con *n*=8. Las diagonales se numeran desde *d*=1 hasta *d*=*n*-1 en el orden en el que tienen que

recorrerse; la diagonal principal *d*=0 corresponde a la inicialización. Cada diagonal tiene *n*-*d* elementos que numeraremos del *i*=1 al *i*=*n*-*d*. Así este índice nos sirve directamente para conocer la fila en la que se encuentra el elemento por el que vamos; la columna podemos calcularla mediante *j*=*i*+*d*. Obsérvese también que el número de la diagonal indica el número de productos de matrices necesarios. Para *d* = 0, solo hay una matriz y ningún producto que hacer; para *d*=*n*-1, con *j*=*n* e *i*=1 (dato buscado) hay *n* matrices y n-1 productos de matrices.

Cuando la tabla esté completamente llena, el número mínimo de multiplicaciones escalares necesarias para multiplicar las *n* matrices será *matrices*[1, n].

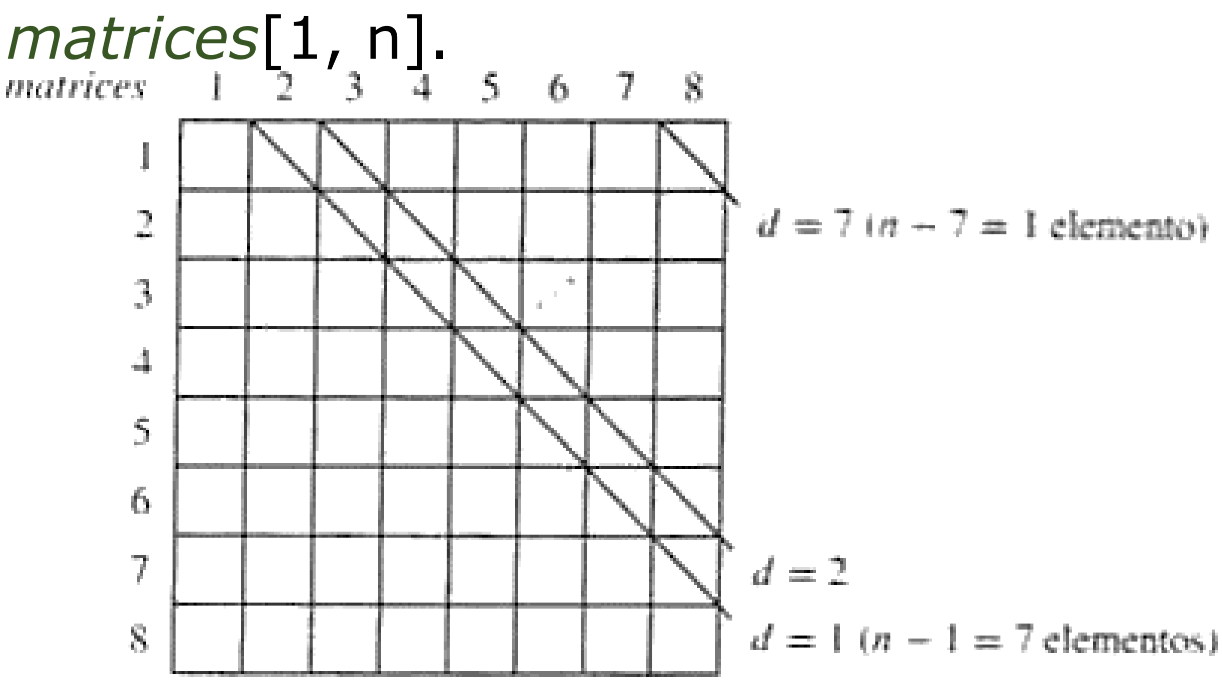


Figura 13.5: Cómo recorrer una tabla por diagonales.

Además de ir calculando la tabla *matrices*, guardaremos en una *tabla de decisiones P[1..n,* 1..n] cómo se van colocando los paréntesis, porque aunque esta información se pueda obtener de la

tabla *matrices* sería más costoso. Así *P[i, j] = k* representa que al considerar las matrices de la *i* a la *j.* hemos dividido así

$$(M_i, \ldots M_k)(M_{k+1} \ldots Mj).$$

mientras que P[i, *j*]= 0 indica que no hacen falta paréntesis, por encontramos en un caso básico *i=j.*

El algoritmo que hace los cálculos es el siguiente:

```
fun multiplicacionMatrices(D[0..n] de nat⁺)
dev ⟨numMin: nat, P[1..n, 1..n] de 0..n⟩
var matrices[1..n, 1..n] de nat∞

{inicialización, diagonal principal}
para i=1 hasta n hacer
   matrices[i, i] := 0;
   P[i,j]:= 0;
fpara

{recorrido por diagonales}
para d=1 hasta n - 1 hacer {recorre diagonales}
   para i = 1 hasta n - d hacer {recorre
   elementos dentro de la diagonal}
      j := i + d;
      {calcular mínimo}
      matrices[i,j] := +∞;
      para k=i hasta j - 1 hacer
         temp := matrices[i, k] + matrices[k+1, j] +
         D[i-1] * D[k]*D[j];
         si temp < matrices[i, j] entonces
            matrices[i,j] := temp;
            P[i,j] := k;
         fsi
      fpara
   fpara
fpara
numMin := matrices[1, n];
ffun
```

El coste en tiempo del algoritmo está en $\Theta(n^3)$ y en espacio adicional en $\Theta(n^2)$.

Para escribir la secuencia de matrices

con paréntesis, teniendo en cuenta que cuando hay menos de tres matrices no hace falta poner paréntesis porque solo hay una forma de hacer el producto, utilizamos el siguiente algoritmo recursivo de coste lineal con respecto al número de matrices.

{1≤i≤j≤n}

```
proc escribirParéntesis (e i, j : nat, e P[1..n,
1..n] de nat)

si i = j entonces
   imprimir("Mi");
si no
   k := P[i,j];
   si k > i entonces
      imprimir(();
      escribirParéntesis(i, k, P);
      imprimir());
   si no
      imprimir("Mi");
   fsi

   si k+1 < j entonces
      imprimir(();
      escribirParéntesis(k+1, j, P);
      imprimir());
   si no
      imprimir("Mj");
   fsi
fsi
fproc
```

13.7. Claves

Sean $c_1 < \ldots < c_n$ un conjunto de claves distintas ordenadas, y sea p_i la probabilidad con que se pide buscar la clave *ci y* su información asociada, para *i* entre 1 y *n.* Se desea encontrar un árbol de búsqueda que minimice el número medio de comparaciones necesarias para realizar una búsqueda, suponiendo que $\sum_{i=1}^{n} p_i = 1$, es decir, que todas las peticiones se refieren a claves que están en el árbol.

-----------------------------Solución------------------------------

Dadas las *n* claves ordenadas, la primera decisión que hay que tomar para construir un árbol de búsqueda es ver qué clave se va a poner en la raíz. Si elegimos para la raíz la clave c_k, por las propiedades de los árboles de búsqueda (véase el Ejercicio 7.1) sabemos que todas las claves $\{c_1,\ldots,c_{k-1}\}$ tienen que ir al hijo izquierdo y todas las claves $\{c_{k+1},\ldots, c_n\}$ tienen que ir al hijo derecho, como muestra la Figura 13.6.

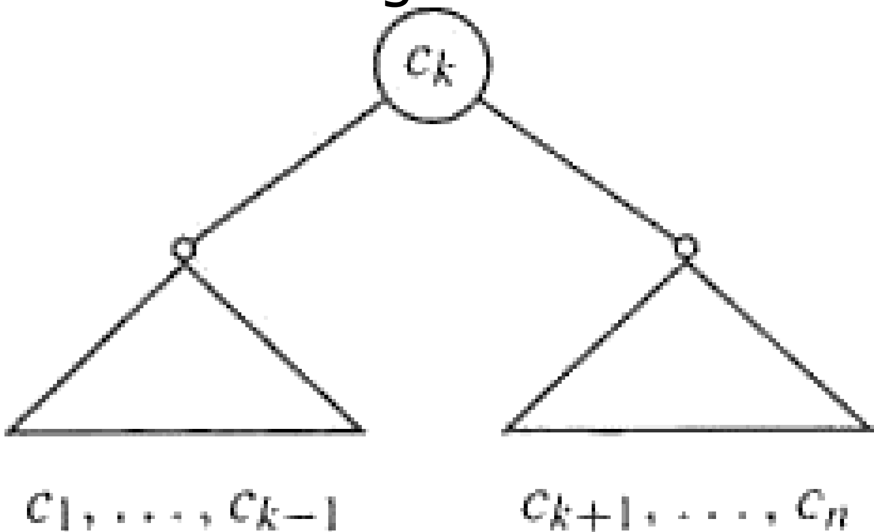


Figura 13.6: Árbol de búsqueda con claves $c_1 < \ldots < c_n$.

Consideremos un árbol de búsqueda *a*

para las claves $\{c_1,\dots,c_n\}$ en el que la clave Q está en la raíz.

Utilizamos la función

compa(1, n) = número medio de comparaciones para realizar una búsqueda en *a* en las condiciones del enunciado.

Lo primero que hace falta notar es que el número de comparaciones para encontrar una clave se corresponde con el nivel que esa clave ocupa en el árbol, empezando a contar por la raíz. En particular, el número de comparaciones para c_k es 1. Por tanto, el número medio de comparaciones viene dado por el siguiente cálculo, donde utilizamos *hi(a) y hd(a)* para denotar el hijo izquierdo y el hijo derecho, respectivamente, del árbol *a*.

$$\begin{aligned}
\mathrm{comp}_a(1,n) &= \sum_{l=1}^{n} p_l \text{ nivel }_a(c_l) \\
&= p_k + \sum_{l=1}^{k-1} p_l \text{ nivel }_a(c_l) + \sum_{l=k+1}^{n} p_l \text{ nivel }_a(c_l) \\
&= p_k + \sum_{l=1}^{k-1} p_l(\text{ nivel }_{hi(a)}(c_l)+1) + \sum_{l=k+1}^{n} p_l(\text{ nivel }_{hd(a)}(c_l)+1) \\
&= p_k + \sum_{l=1}^{k-1} p_l + \sum_{l=k+1}^{n} p_l + \sum_{l=1}^{k-1} p_l \text{ nivel }_{hi(a)}(c_l) + \sum_{l=k+1}^{n} p_l \text{ nivel }_{hd(a)}(c_l) \\
&= \sum_{l=1}^{n} p_l + \mathrm{comp}_{hi(a)}(1,k-1) + \mathrm{comp}_{hd(a)}(k+1,n)
\end{aligned}$$

Para que el valor en el lado izquierdo de la ecuación sea mínimo, también han de serlo los dos de la derecha, por ser independientes entre sí, es decir, se cumple el principio de optimalidad: para que el árbol *a* sea óptimo, sus dos hijos tienen que serlo también.

Naturalmente, la expresión anterior se obtiene suponiendo que la clave c_k está

en la raíz. Como esto a priori no lo sabemos, necesitaremos calcular el *mínimo,* al variar *k* entre 1 y *n* (en el caso inicial, que enseguida vamos a generalizar). Es decir, hay que probar todas las posibilidades de colocar una clave en la raíz.

Notemos que. para poder después hacer la generalización adecuada (de i a *j* en vez de 1 a *n*). conviene dejar la suma de las probabilidades tal como está, pues la información $\sum_{l=1}^{n} p_l = 1$ no es generalizable.

Antes de plantear la definición recursiva que nos interesa, vamos a hacer un pequeño ajuste sobre los índices. Como el índice *k* de la clave que se elige para la raíz varía de 1 a *n.* cuando *k* vale 1 o *n* se obtienen casos básicos (1, *k*-1) = (1,0) en el hijo izquierdo, *o (k*+1, n) = (n+1, n) en el hijo derecho. Así el primer índice variaría de 1 a *n*+1 y el segundo de 0 a *n.* Para hacer que los dos índices tengan el mismo rango y facilitar el recorrido posterior por diagonales, vamos a restar 1 al primero, con lo cual la definición queda como sigue:

comp(i, j) = número medio *mínimo* de comparaciones en un árbol de búsqueda conteniendo las claves $c_{i+1},\ldots,c_j$.

El caso que nos interesa (la llamada inicial, desde el punto de vista recursivo), para un árbol con las *n* claves, será

comp(0, n), definido por

$$\text{comp}(0,n) = \sum_{l=1}^{n} p_l + \min_{1\le k\le n}\{\text{comp}(0,k-1) + \text{comp}(k,n)\}.$$

(La suma de probabilidades puede sacarse fuera del mínimo porque no depende del índice k, es decir, es constante con respecto a k.)

En el caso general, la recurrencia se define para *0≤i≤j≤n:*

$$\text{comp}(i,j) = \sum_{l=i+1}^{j} p_l + \min_{i+1\le k\le j}\{\text{comp}(i,k-1) + \text{comp}(k,j)\}$$

El caso básico es *comp(i, i)* = 0. donde 0≤*i*≤*n,* que corresponde al árbol vacío.

Vamos a rellenar por diagonales una matriz *comp[0..n,* 0..n], como hemos hecho en la solución del Ejercicio 13.6, utilizando para ello un índice *d* que recorra las diagonales, y un índice *i* que recorra los elementos dentro de la diagonal *d.* En este caso, la matriz es de dimensiones (n+1) x *(n*+1) por lo que la diagonal *d*=0 tiene *n*+1 elementos, la diagonal *d=1* tiene *n* elementos, y así sucesivamente; en general, la diagonal *d* tiene *n*-*d*+1 elementos, que numeraremos desde *i*=0 hasta *i*=*n*-*d.* La columna en la que se encuentra el elemento *i* de la diagonal *d* es *i*+*d.*

Utilizaremos una matriz adicional *prob[0..n,* 0..n] para ir calculando las sumas de probabilidades que necesitamos para calcular incrementalmente cada valor de la matriz *comp.* De no hacerlo

así. el coste del algoritmo aumentaría. Con la definición

$$\text{prob}[i,j] = \sum_{l=i+1}^{j} p_l$$

se puede calcular la matriz mediante la fórmula

prob[i, j] = prob[I, j-1] + *pj.*

teniendo como caso básico *prob[i, i*] = 0. Como cada entrada en esta matriz se calcula a partir de otra en la misma fila y una columna anterior, o sea. en la diagonal anterior, podemos utilizar el mismo bucle principal para calcular tanto *comp* como *prob.*

Por último guardaremos en una tercera matriz raíz[0..*n*, 0..*n*] las decisiones que vamos tornando sobre las raíces de los árboles óptimos (o sea, en qué índice *k* se obtiene el mínimo), de forma que raíz[i, j] será la raíz del árbol de búsqueda óptimo con las claves $c_{i+1},\ldots,c_j$. El caso básico, correspondiente al árbol vacío, lo vamos a representar en esta matriz con el valor 0.

El algoritmo completo es como sigue:

{C[1]< … <C[n] ∧ ∀i: 1≤i≤n :0≤P[i]≤1}

```
fun arbolBusquedaOptimo(C[1..n] de clave, P[1..n] de real) dev ⟨númComp: real, raíz[0..n, 0..n] de 0..n⟩
var comp[0..n, 0..n] de real∞, prob[0..n, 0..n] de real

  para i = 0 hasta n hacer
      comp[i,i]:= 0;
      prob[i,i] := 0;
      raíz[i,i]:= 0;
```

```
fpara

para d=1 hasta n hacer {recorre diagonales}
  para i=0 hasta n-d hacer {recorre elementos dentro de la diagonal}
    j:= i+d;
    prob[i,j]:= prob[i, j-1] + P[j];
    {calcular mínimo}
    comp[i, j]:= +∞;
    para k=i+1 hasta j hacer
      temp:= comp[i,k- 1] + comp[k, j];
      si temp < comp[i, j] entonces
        comp[i, j]:= temp;
        raíz[i, j]:= k;
      fsi
    fpara
    comp[i, j] := comp[i, j]+ prob[i, j];
  fpara
fpara
númComp:= comp[0,n];
ffun
```

El COSTE de este algoritmo en espacio adicional está en $\Theta(n^2)$ debido a las 2 matrices *comp* y *prob.*

El COSTE en tiempo está en $\Theta(n^3)$, debido a los 3 bucles anidados: los dos primeros para recorrer la (mitad superior de la) matriz y el tercero para calcular el mínimo.

Finalmente, el algoritmo recursivo que construye el árbol de búsqueda óptimo, a partir del vector de claves y de las decisiones sobre las raíces guardadas por el algoritmo anterior en la matriz *raíz,* cuya llamada inicial será construirArbol(C, *raíz.* 0, n), es el siguiente:

(C[1] < … < C[n] ∧ *i*≤*j* }

```
fun construirArbol(C[1..n] de clave,
raíz[0..n, 0..n] de 0..n, i, j: 0..n) dev árbol:
árbol-bb[clave]
var iz, dr: árbol-bb[clave]

si raíz[i, j] = 0 entonces
  árbol := abb-vacío();
si no
  k:= raíz[i,j];
  iz:= construirArbol(C, raíz, i, k-1);
  dr:= construirArbol(C, raíz, k, j);
  árbol:= plantar(iz, C[k], dr);
fsi
ffun
```

Una vez que todas las decisiones con respecto a la definición del árbol ya han sido tomadas mediante la función árbolBúsquedaOptimo, y suponiendo que el coste de las operaciones constructoras de árboles binarios es constante (como es el caso si se implementan mediante punteros, véase el Ejercicio 7.2), el coste de construirArbol(C, *raíz*, 0, *n)* es lineal con respecto a *n*, que es el número de nodos en el árbol, puesto que el proceso de construcción pasa una vez por cada clave/nodo y una vez por cada hijo vacío.

13.8. Multiplicar letras

Sea un alfabeto X = {n, *b,* c) con la siguiente "tabla de multiplicación" (donde cada fila corresponde al símbolo izquierdo y cada columna al derecho; por ejemplo, *ab = b.ba = c.* etc.):

	a	*b*	*c*
a	*b*	*b*	*a*
b	*c*	*b*	*a*
c	*a*	*c*	*c*

Nótese que dicha multiplicación no es asociativa ni conmutativa.

(a) Escribir un algoritmo que dada una cadena $x = x_1 \ldots x_n$ de caracteres de Σ determine si es posible insertar paréntesis en x de forma que el valor de la expresión resultante sea *a.* Por ejemplo, si x = *bbbba,* el algoritmo debe devolver SÍ dado que *(b(bb))(ba) = (bb)c = bc = a.*

(b) Escribir otro algoritmo para calcular el número de formas diferentes de insertar paréntesis en x para que el resultado sea *a.*

(c) Generalizar el problema de forma que la tabla de multiplicación sea un parámetro **adicional.**

------------------------Solución-----------------

Apartado (a)

Para saber si es posible poner paréntesis en $x_1 \ldots x_n$ de forma que su valor resulte ser *a* consideramos la posición *k* en la que se insertan los paréntesis más externos

$$(x_1 \ldots x_k)(x_{k+1} \ldots x_n)$$

y observamos la tabla de multiplicación para ver las formas de conseguir el resultado *a*:

$$a = ac = bc = ca.$$

Esto significa que para resolver recursivamente el problema necesitamos saber no sólamente si es posible insertar paréntesis en $x_1 \ldots x_k$ o $x_{k+1} \ldots x_n$ para obtener *a,* sino también para obtener *b y* c.

Ello nos lleva a considerar los 3 predicados siguientes:

- *A(i, j)* que indica si es posible insertar paréntesis en $x_i \ldots x_j$ para obtener *a,*
- *B(i, j)* que indica si es posible insertar paréntesis en $x_i \ldots x_j$ para obtener *b.*
- *C(i, j)* que indica si es posible insertar paréntesis en $x_i \ldots x_j$ para obtener *c.*

Los mismos pueden ser definidos de forma mutuamente recursiva como sigue, teniendo en cuenta que la posición *k* en la que se insertan los paréntesis más externos en $x_1, \ldots, x_j$ puede variar entre *i y j*-1, con *j* > *i.*

$$A(i,j) = \bigvee_{k=i}^{j-1} ((A(i,k) \wedge C(k+1,j))$$
$$\vee (B(i,k) \wedge C(k+1,j))$$
$$\vee (C(i,k) \wedge A(k+1,j)))$$

$$B(i,j) = \bigvee_{k=i}^{j-1} ((A(i,k) \wedge A(k+1,j))$$
$$\vee (A(i,k) \wedge B(k+1,j))$$
$$\vee (B(i,k) \wedge B(k+1,j)))$$

$$C(i,j) = \bigvee_{k=i}^{j-1} ((B(i,k) \wedge A(k+1,j))$$
$$\vee (C(i,k) \wedge B(k+1,j))$$
$$\vee (C(i,k) \wedge C(k+1,j)))$$

Los casos básicos se obtienen cuando *i*= *j*. En tal caso basta comparar el carácter xi con el buscado:
A(i, i) = (xi=a)
B(i. i) = (xi=b)
C(i, i) = (xi=c)

Como en el Ejercicio 13.6, la forma de resolver el problema iterativamente consiste en rellenar una tabla por diagonales, empezando por la diagonal principal para el caso básico. Como tenemos 3 funciones mutuamente recursivas, necesitamos considerar 3 matrices *n x n* (o equivalentemente una matriz tridimensional donde la tercera dimensión va de *a* a *c*) que se rellenan simultáneamente.

Vamos a completar el algoritmo de manera que también se devuelva una tabla de decisiones que nos permita calcular la forma adecuada de insertar los paréntesis. Cada decisión consistirá en la posición *k* en la que se divide, y los caracteres que hay que obtener en cada parte. Como tenemos tres predicados, utilizaremos 3 matrices correspondientes de decisiones.

Observemos finalmente que, como el tercer bucle (para *k)* tiene que calcular una disyunción múltiple, puede parar tan

pronto se obtenga un factor cierto. Utilizamos un procedimiento auxiliar llamado asignar para dar valor a los 3 campos de un registo de tipo *decisión*.

```
tipos
  decisión = reg
        posición : nat
        resultado₁, resultado₂ : car
      freg
ftipos
```

```
fun paréntesisA(X[1..n de car) dev ⟨posible :
bool, pa[1..n. 1..n], pb[1..n, 1..n], pc[1..n, 1..n]
de decisión⟩
var A[1..n, 1..n], B[1..n, 1..n], C[1..n, 1..n] de
bool

para d=1 hasta n hacer
  A[i, i] := (X[i] = a);
  B[i, i] := (X[i] = b);
  C[i, i] := (X[i] = c);
fpara

para d=1 hasta n - 1 hacer {recorre diagonales}
 para i=1 hasta n - d hacer {recorre elementos
 dentro de la diagonal}
  j := i + d;
  A[i, j] := falso;
  k := i;
  {calcula disyunción para A}
  mientras ¬A[i, j] ∧ k<j hacer
    si A[i, k] ∧ C[k+1, j] entonces
      A[i, j] := cierto;
      asignar(pa[i, j], k, a, c);
    si no
      si B[i, k] ∧ C[k+1, j] entonces
        A[i, j] := cierto;
        asignar(pa[i, j], k, b, c);
      sino
        si C[i,k] ∧ A[k+1, j] entonces
          A[i, j] := cierto;
```

```
        asignar(pa[i, j], k, c, a);
      si no
        k := k+1;
      fsi
    fsi
  fsi
fmientras
B[I, j] := falso;
k := i;
{calcula disyunción para B}
mientras ⌐B[i, j] ∧ k<j hacer
  si A[i, k] ∧ A[k+1, j] entonces
    B[i, j] := cierto;
    asignar(pb[i,j], k, a, a);
  si no
    si A[i, k] ∧ B[k+1, j] entonces
      B[i, j] := cierto;
      asignar(pb[i, j], k, a, b);
    sino
      si B[i, k] ∧ B(k+1, j] entonces
        B[i,j] := cierto;
        asignar(pb[i, j], k, b, b);
      si no
        k := k+1;
      fsi
    fsi
  fsi
fmientras
C[i, j] := falso;
k := i;
{calcula disyunción para C}
mientras ⌐C[i, j] ∧ k<j hacer
  si B[i, k] ∧ A[k+1, j] entonces
    C[i, j] := cierto;
    asignar(pc[i, j], k, b, a);
  si no
    si C[i,k] ∧ B[k+1, j] entonces
      C[i, j] := cierto;
      asignar(pc[i, j], k, c, b);
    si no
      si C[i, k] ∧ C[k + 1.71 entonces
        C[i, j] := cierto;
        asignar(pc[i, j], k, c, c);
      si no
        k := k+1;
```

```
          fsi
        fsi
      fsi
    fmientras
  fpara
fpara
posible := A[1,n];
ffun
```

```
proc asignar(d: decisión, e p: nat, e r₁, r₂: car)

d.posición := p;
d.resultado₁ := r₁;
d.resultado₂ := r₂;
fproc
```

Las matrices de decisiones *pa, pb y pc* solamente se consultan cuando *posible* = cierto. En tal caso, si *i* < *j*, la información *px[i, j]* = ⟨*x*, *y*, z⟩ significa que hay que poner paréntesis en la posición *k* y a continuación recursivamente consultar *py*[i, k] y *pz*[k+1,j] (donde *x*, *y* y *z* pueden ser *a*, *b* ó *c*). El caso básico se tiene cuando *i* = *j*, indicando que solo hay un carácter y por tanto no hace falta poner paréntesis.

El coste de este algoritmo está en $\Theta(n^3)$ debido a los tres bucles anidados, mientras que su coste en espacio auxiliar está en $\Theta(n^2)$. Nótese que manejar *3* matrices en vez de una no afecta al coste en espacio, por tratarse de una constante.

Apartado (b)-------------------

La estructura de la solución para esta segunda parte es completamente análoga, cambiando las operaciones booleanas de conjunción y disyunción por operaciones aritméticas de producto y suma, respectivamente. Manteniendo la misma notación se obtienen las ecuaciones siguientes:

A (i, j) =número de formas de insertar paréntesis en $x_1 \ldots x_j$ para obtener *a*.

B(i, j) =número de formas de insertar paréntesis en $x_1 \ldots x_j$ para obtener *b*.

C(i, j) =número de formas de insertar paréntesis en $x_1 \ldots x_j$ para obtener *c*.

$$A(i,j) = \sum_{k=i}^{j-1} ((A(i,k) * C(k+1,j)) + (B(i,k) * C(k+1,j)) + (C(i,k) * A(k+1,j)))$$

$$B(i,j) = \sum_{k=i}^{j-1} ((A(i,k) * A(k+1,j)) + (A(i,k) * B(k+1,j)) + (B(i,k) * B(k+1,j)))$$

$$C(i,j) = \sum_{k=i}^{j-1} ((B(i,k) * A(k+1,j)) + (C(i,k) * B(k+1,j)) + (C(i,k) * C(k+1,j)))$$

A(i, i) = si (x_i = *a*) entonces 1, si no 0

B(i, i) = si (x_i = b) entonces 1, si no 0
C(i, i) = si (x_i = c) entonces 1, si no 0

Como en este caso no hay que tomar decisiones, el algoritmo se simplifica un poco, y los 3 bucles "**mientras**" para *k* se pueden juntar en uno único. Los costes son los mismos que en el apartado anterior.

```
fun formas-paréntesis-a(X[1..n] de car) dev
número: nat
var A[1..n, 1..n], B[1..n, 1..n], C[1..N, 1..n] de
nat

para i = 1 hasta n hacer
    si X[i]=a entonces A[i,i]:=1 sino A[i,i]:= 0 fsi
    si X[i]=b entonces B[i,i]:=1 sino B[i,i]:= 0 fsi
    si X[i]=c entonces C[i,i]:=1 sino C[i,i]:= 0 fsi
fpara

para d=1 hasta n-1 hacer {recorre diagonales}
    para i=1 hasta n-d hacer {recorre elementos
    dentro de la diagonal}
        j := i + d;
        A[i,j] := 0;
        B[i,j] := 0;
        C[i, j] := 0;
        {calcula sumatorios}
        para k= i hasta i+ d-1 hacer
            A[i, j]:= A[i, j] + A[i,k] * C[k+1, j] +
            B[i, k] * C[k+1, j] + C[i, k] * A[k+1, j];
            B[i,j] := B[i,j] + A[í,k]*A[k+1,j] +
            A[i,k]*B[k+1, j] + B[i, k] * B[k+1, j];
            C[i, j]:= C[i, j] + B[i, k] * A[k+1, j] +
            C[i, k] * B[k+1, j] + C[i, k] * C[k+1, j];
        fpara
    fpara
fpara
número := A[1,n];
ffun
```

Apartado (c)-------------------------------

Los algoritmos anteriores tienen la pega

de que su estructura depende de la tabla del enunciado. Vamos a generalizarlos de forma que la tabla sea otro parámetro adicional. Dado un conjunto de *m* elementos indexados adecuadamente. $e_1,\ldots,e_m$, nos dan una tabla de multiplicación arbitraria T(1..m, 1..m] de forma que T[c, f] = *r* indica que el resultado de multiplicar los elementos e_c *y* e_f es el elemento e_r. Vamos a suponer que la cadena X[1..n] es de números naturales de 1 a *m,* representando la cadena de elementos $e_{X[1]}\ldots e_{X[n]}$.

Como no tenemos información a priori de la tabla, en el algoritmo nos vemos obligados a recorrer toda la tabla para encontrar cada elemento como producto de otros dos. Para simplificar el algoritmo, no vamos a intentar optimizar parando tan pronto como encontremos una posible forma de poner paréntesis, como se hacía en el algoritmo paréntesis-a del Apartado (a). Además, calcularemos en el mismo algoritmo el número de formas de poner paréntesis. Al efecto, evitaremos la suma de sumandos nulos, a diferencia de lo que sucedía en el algoritmo anterior formas-paréntesis-a en el que sumábamos todos los sumandos.

Debido a que *m* forma parte de los datos de entrada del algoritmo, utilizaremos tablas *tridimensionales P[* 1..n, 1*..n,* 1..m], N[1..n, 1..n, 1..m], D[1..n, 1..n, 1..m] de modo que P[i, j, r] indica si es posible o no poner paréntesis

en la cadena X[i..j] para obtener el elemento e_r, N[i, j, r] nos dice el número de formas posibles, y en la tabla de decisiones D[i, j, r] = ⟨*k,c, f*⟩ significa que una forma posible de lograrlo es poner paréntesis en la posición *k* y a continuación recursivamente consultar *D[i, k, c*] y D[k+1, j, f] hasta llegar a un caso básico (por supuesto, *D[i, j,* r] solamente debe consultarse cuando P[i, j, r] = cierto).

Finalmente, nótese que el argumento a del algoritmo siguiente es un número entre 1 y *m*. En concreto, buscamos poner paréntesis en la cadena $e_{X[1]}…e_{X[n]}$ para obtener e_a.

```
fun paréntesisGeneral(T[1..m, 1..m] de 1..m,
X[1..n] de 1..m, a: 1..m)
dev ⟨posible: bool, número: nat, D[1..n, 1..n,
1..m] de decisión⟩
var P[1..n, 1..n, 1..m] de bool, N[1..n, 1..n,
1..m] de nat

{inicializa tablas}
para i = 1 hasta n hacer
  para j = 1 hasta n hacer
    para k = 1 hasta m hacer
      P[i, j, k] := falso;
      N[i, j, k] := 0;
    fpara
  fpara
  P[i, i, X[i]] := cierto;
  N[i, i, X[i]] := 1;
fpara

{rellena tablas por diagonales olvidando la tercera
dimensión}
para d = 1 hasta n - 1 hacer
```

```
para i = 1 hasta n - d hacer
  j := i + d;
  para k = i hasta j - 1 hacer
    { recorre tabla de multiplicar T }
    para f = 1 hasta m hacer
      para c = 1 hasta m hacer
        si P[i,k,f] ∧ P[k+1, j, c] entonces
          r := T[f, c];
          P[i, j, r] := cierto;
          asignar(D[i, j, r], k, c, f);
          N[i, j, r] := N[i, j, r] + N[i, k, f] * N[k+1, j,
          c];
        fsi
      fpara
    fpara
  fpara
fpara
fpara
posible := P[1,n,a];
número := N[1,n,a];
ffun
```

Este algoritmo utiliza el procedimiento asignar del Apartado (a), pero hay que ajustar sus tipos apropiadamente debido a la indexación de los caracteres que se usa en este apartado.

El coste en tiempo de este algoritmo está en $\Theta(n^3m^2)$ debido a los cinco bucles anidados, y su coste en espacio auxiliar está en $\Theta(n^2m)$. Para una tabla fija *T*, *m* se convierte en constante, de modo que. por ejemplo, para la tabla del enunciado con *m*=3 tenemos el mismo coste que en la versión anterior.

13.9. Caminos de coste minimo

Dado un grafo dirigido cuyas aristas están valoradas con números positivos, desarrollar algoritmos para

(a) calcular el ***coste*** *(del camino)* *mínimo* entre cada par de vértices del grafo.

(b) calcular el *número* de caminos de coste mínimo para cada par de vértices del grafo, y

(c) calcular la *clausura reflexiva y transitiva* de la relación dada por el grafo.

-----------------------------Solución:--------------------------

En los 2 primeros apartados supondremos que el grafo viene dado por su matriz de valores (véase el Ejercicio 9.6) con $+\infty$ como el valor especial para indicar que no existe una arista.

Apartado (a)---------------------

La solución que presentamos en este apartado se conoce como *algoritmo de* ***Floyd*** y se basa en la siguiente función (para $0 \le k \le n$ y $1 \le i, j \le n$) que en principio definiremos recursivamente:

$C^k(j, j)$ = coste *mínimo* para ir de *i* a *j* pudiendo utilizar como vértices intermedios aquellos entre 1 y *k*.

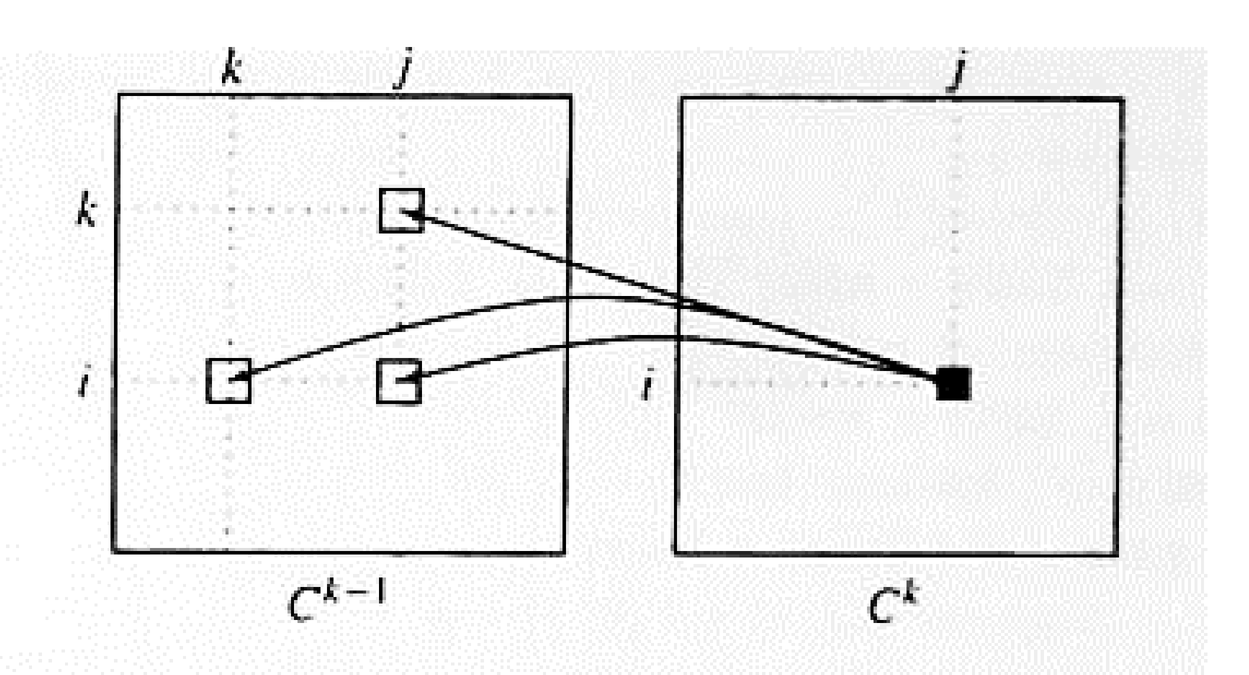


Figura 13.7: Esquema de tabla para el algoritmo de Floyd.

Para calcular el camino de coste mínimo entre los vértices i y *j* podemos considerar 2 posibilidades:

o bien no pasamos por el vértice *k,* en cuyo caso tendremos que calcular el mejor camino con el resto de los vértices. $C^{k-1}(i, j)$;

o bien sí pasamos por el vértice *k,* en cuyo caso tendremos que obtener caminos que vayan de *i* a *k* y de *k* a *j.* Como el coste del camino completo se obtiene sumando los costes de los caminos parciales, para que el primero sea óptimo también han de serlo los segundos, es decir, se cumple el *principio de optimalidad*. Ninguno de estos caminos parciales utilizará *k* como vértice intermedio, ya que en ese caso habría algún ciclo que podríamos eliminar, mejorando el coste.

La recurrencia, con $1 \le k, \ i, j \le n$, es

$$C^k(i,j) = min\{C^{k-1}(i,j), C^{k-1}(i,k) + C^{k-1}(k,j)\}.$$

El caso básico se presenta cuando k=0 en cuyo caso solo existe la posibilidad de ir directamente de *i* a *j,* con el coste de la

correspondiente arista si los dos vértices no coinciden, o con coste nulo si se trata del mismo vértice:

$$C^0(i,j) = \begin{cases} G[i,j] & \text{si } i \neq j \\ 0 & \text{si } i = j \end{cases}$$

El coste mínimo entre i y *j* será $C^n(i, j)$.

La recurrencia contiene 3 argumentos: *i, j* y *k,* por lo que al aplicar el método de programación dinámica en principio necesitaríamos *n*+1 matrices de dimensiones *n* x *n.* con un espacio adicional en $\Theta(n^3)$. Pero afortunadamente podemos reducir este espacio adicional.

En primer lugar, notemos que para calcular la matriz C^k solo necesitamos la matriz C^{k-1}. Veamos ahora que las actualizaciones se pueden ir realizando sobre la misma matriz. Para calcular la posición *(i, j)* de la matriz C^k necesitamos 3 posiciones de la matriz C^{k-1}, como muestra la Figura 13.7. Pero la fila *k* y la columna *k* no cambian cuando hacemos la actualización de C^{k-1} a C^k. Para la fila *k* tenemos

$$C^k(k,j) = min\{C^{k-1}(k,j), C^{k-1}(k,k) + C^{k-1}(k,j)\} = C^{k-1}(k,j).$$

ya que $C^{k-1}(k, k) = 0$ (es fácil ver por inducción sobre *j* que $C^j(k, k) = 0$ para todo *j* entre 0 y n). Y de igual forma $C^k(i, k) = C^{k-1}(i, k)$ para la columna *k.*

Por tanto, podemos limitarnos a utilizar solo una matriz C[1..n, 1..n] en la que finalmente devolvemos la solución al

problema, de modo que el coste en espacio adicional está en Θ(1).

Por otra parte, en una matriz auxiliar *camino*[1..n, 1..n] guardaremos información sobre los caminos óptimos: *camino[i,* j] = *k* ≠ 0 indica que el camino de coste mínimo de *i* a *j* pasa por *k,* mientras que *camino[i, j]* = 0 indica que se va directamente de i a *j.*

El algoritmo, cuyo coste en tiempo de ejecución está en $\Theta(n^3)$, es el siguiente:

```
fun Floyd(G: grafo-val[n]) dev (C[1..n, 1..n] de
real∞, camino[1..n, 1..n] de 0..n)

{inicialización}
C:= G;
camino[1..n, 1..n] := [0];
para i = 1 hasta n hacer
    C[i, i]:= 0;
fpara

{actualizaciones de la matriz}
para k = 1 hasta n hacer
  para i = 1 hasta n hacer
    para j = 1 hasta n hacer
      temp := C[i, k] + C[k, j];
      si temp < C[i, j] entonces
        C[i, j] := temp;
        camino[i, j]:= k;
      fsi
    fpara
  fpara
fpara
ffun
```

El algoritmo que imprime el camino mínimo entre cada par de vértices, utilizando la matriz *camino* calculada anteriormente, es el siguiente, (que utiliza un procedimiento recursivo auxiliar

imprCaminoAux).

```
proc imprimirCaminos(e C[1..n, 1..n] de
real∞, e camino[1..n, 1..n] de 0..n)

para i = 1 hasta n hacer
  para j = 1 hasta n hacer
    si C[i, j] < +∞ entonces
      imprimir("camino de", i, "a", j);
      imprimir(i);
      imprCaminoAux(i, j, camino);
      imprimir(j);
    fsi
  fpara
fpara
fproc
```

```
proc imprCaminoAux(e i, j: 1..n, e
camino[1..n, 1..n] de 0..n)

k := camino[i, j];
si k > 0 entonces {hay un camino no directo}
   imprCaminoAux(i, k, camino);
   imprimir(k);
   imprCaminoAux(k, j, camino);
fsi
fproc
```

Apartado (b)--------------------------------

Tenemos que ir calculando a la vez el coste del camino mínimo (apartado anterior) y el número de caminos mínimos. Definimos la siguiente función para $0 \leq k \leq n$ y $1 \leq i, j \leq n$:

$ncm^k(i, j)$ = número de caminos de coste mínimo de *i* a *j* utilizando como posibles vértices intermedios aquellos entre 1 y k.

Si no hay caminos de coste mínimo de *i* a *j* que pasen por k, bien porque $C^{k-1}(i, j) < C^{k-1}(i, k) + C^{k-1}(k, j)$ o porque i=k ó j=k, hay que calcular el número de caminos de coste mínimo sin pasar por k. Si todos los caminos de coste mínimo pasan por k, lo que sucede cuando $C^{k-1}(i, j) > C^{k-1}(i, k) + C^{k-1}(k, j)$, los posibles caminos de *i* a *j* serán todas las combinaciones de caminos óptimos de *i* a k con caminos óptimos de k a *j.* Finalmente, si cuesta lo mismo pasar por k que no pasar, es decir, cuando $C^{k-1}(i, j) = C^{k-1}(i, k) + C^{k-1}(k, j)$, tendremos que sumar todas las posibilidades. La recurrencia, por tanto, es la siguiente:

$$ncm^k(i,j) = \begin{cases} ncm^{k-1}(i,j) \text{si} (C^{k-1}(i,j) < C^{k-1}(i,k) + C^{k-1}(k,j)) \vee i = k \vee j = k \\ ncm^{k-1}(i,k) * ncm^{k-1}(k,j) \text{si } C^{k-1}(i,j) > C^{k-1}(i,k) + C^{k-1}(k,j) \\ ncm^{k-1}(i,j) + ncm^{k-1}(i,k) * ncm^{k-1}(k,j) \\ \text{si} (C^{k-1}(i,j) = C^{k-1}(i,k) + C^{k-1}(k,j)) \wedge i \neq k \wedge j \neq k \end{cases}$$

Y el caso básico es

$$n^0(i,j) = \begin{cases} 0 & \text{si } G[i,j] = +\infty \wedge i \neq j \\ 1 & \text{si } G[i,j] \neq +\infty \vee i = j. \end{cases}$$

Podemos utilizar una única matriz que vamos actualizando ya que cuando calculamos la posición (i, j) de la matriz ncm^k solo necesitamos, además de ncm^{k-}

[1](i, j), una posición en la fila *k* y otra en la columna *k,* pero por el primer caso de la recurrencia, cuando *i=k* ó *j=k* la matriz no cambia.

El algoritmo, de coste en tiempo en $\Theta(n^3)$ y en espacio adicional en $\Theta(n^2)$, es el siguiente:

```
fun númeroCaminosMínimos(G: grafoVal[n])
dev ncm[1..n, 1..n] de nat
var C[1..n, 1..n] de real∞
{inicialización}
para i = 1 hasta n hacer
  para j= 1 hasta n hacer
    C[i,j] := G[i, j];
    si G[i,j] = +∞ entonces
      ncm[i,j] := 0;
    si no
      ncm[i, j] := 1;
    fsi
  fpara
fpara

para i = 1 hasta n hacer
  C[i,i] := 0;
  ncm[i,j] := 1;
fpara

{rellenamos simultáneamente las matrices C y ncm}
para k = 1 hasta n hacer
 para i = 1 hasta n hacer
  para j = 1 hasta n hacer
   temp := C[i,k] + C[k,j];
   casos
      temp > C[i,j] v i=k v j=k → nada;
   ▯ temp = C[i,j] ∧ i≠ k ∧ j≠k → ncm[i,j] :=
   ncm[i,j] + ncm[i,k] * ncm[k,j];
   ▯ temp < C[i,j] → C[i,j] := temp; ncm[i,j] :=
   ncm[i,k] * ncm[k,j];
   fcasos
  fpara
 fpara
fpara
ffun
```

Apartado (c)-----------------------------

La relación determinada por un grafo valorado se puede representar mediante una matriz de adyacencia *R* que se obtiene a partir de la matriz de valores G de la forma siguiente:

$$R[i,j] = (G[i,j] \neq +\infty).$$

La clausura reflexiva y transitiva de esta relación coincide con la relación de accesibilidad especificada en el Ejercicio 9.2. El algoritmo que vamos a ver para calcular dicha relación se conoce como *algoritmo de Warshall*. Para ello consideramos una función semejante a la definida para el algoritmo de Floyd,

> $W^k(i, j)$ = booleano que indica si existe un camino de *i* a *j* utilizando como posibles vértices intermedios aquellos entre 1 y A.

que definimos recursivamente de la siguiente manera (para $1 \leq k,i,j \leq n$):

$$W^k(i,j) = W^{k-1}(i,j) \vee \left(W^{k-1}(i,k) \wedge W^{k-1}(k,j)\right)$$

Para ver si podemos ir de *i* a *j* pasando por algunos de los vértices del 1 al *k* vemos si podemos ir de *i* a *j* pasando por algunos de los vértices del 1 al *k*-1, o si podemos ir de *i* a *k* y de *k* a *j* utilizando como vértices intermedios aquellos entre el 1 y el *k*-1, ya que los ciclos no aportan información.

El caso básico se presenta cuando no podemos pasar por ningún vértice intermedio porque *k*=0. En tal caso, para ver si podemos ir de *i* a *j* hay que ver si

podemos hacerlo directamente, es decir, si hay una arista de *i* a *j*. Esa información la tenemos en la matriz de adyacencia dada:

$$W^0(i,j) = R[i,j] \vee i = j$$

Podemos optimizar el espacio utilizando una única matriz que vamos actualizando, ya que. como en el algoritmo de Floyd del Apartado (a), la fila *k* y la columna *k* no cambian al pasar de W^{k-1} a W^k. Para la fila *k* tenemos

$$W^k(k,j) = W^{k-1}(k,j) \vee \left(W^{k-1}(k,k) \wedge W^{k-1}(k,j)\right) = W^{k-1}(k,j).$$

pues $W^{k-1}(k, k)$ = cierto. Lo mismo ocurre para la columna *k*.

El algoritmo es el siguiente:

```
fun Warshall(R: grafo[n]) dev W[1..n, 1..n] de bool

{inicialización}
W := R;
para i = 1 hasta n hacer
    W[i, i] := cierto;
fpara

{actualizaciones}
para k = 1 hasta n hacer
    para i = 1 hasta n hacer
        para j = 1 hasta n hacer
            W[i,j] := W[i,j] ∨ (W[i,k] ∧ W[k,j]);
        fpara
    fpara
fpara
ffun
```

El coste en tiempo de ejecución está en $\Theta(n^3)$ y en espacio adicional en $\Theta(1)$.

13.10. Camino de anchura maxima

Un archipiélago consta de unas cuantas islas y varios puentes que unen ciertos pares de islas entre sí. Para cada puente (que puede ser de dirección única), además de saber la isla origen y la isla destino, se conoce su ***anchura*** > 0.

La ***anchura de un camino*** formado por una sucesión de puentes es la anchura *mínima* entre las anchuras de todos los puentes que lo forman.

Para cada par de islas se desea saber el ***camino de anchura máxima*** que las une (siempre que exista alguno).

----------------------------Solución:---------------------------

Se puede interpretar el archipiélago como un grafo dirigido y valorado, donde los vértices corresponden a las islas y las aristas a los puentes entre ellas. Cada arista está valorada con la anchura del puente que representa. Para un par de islas *i* y *j* conviene representar la anchura de un puente no existente de *i* a *j* utilizando el valor -∞ (elemento neutro del máximo) si *i* i≠ *j,* y el valor +∞ (elemento neutro del mínimo) si *i* = *j,* por lo que supondremos que la matriz de valores del grafo con la información sobre los puentes existentes entre islas viene dada en una matriz *anchura*[1..n, 1..n] tal que

$$\text{anchura}\,[i,j] = \begin{cases} -\infty & \text{si no hay puente de } i \text{ a } j \\ \text{anchura} & \text{si hay puente de } i \text{ a } j \\ +\infty & \text{si } i = j \end{cases}$$

El problema es análogo al problema de los caminos de coste mínimo (resuelto mediante el algoritmo de Floyd en el Apartado (a) del Ejercicio 13.9), en el sentido de que tenemos que buscar caminos óptimos entre todo par de islas. Basándonos en el algoritmo de Floyd definimos una función (para 0≤<≤ *k* ≤< *n* y 1≤i,*j* ≤*n)*

> $A^k(i, j)$ = *anchura máxima* de entre todos los caminos que van de *i* a *j*, pudiendo utilizar como posibles islas intermedias las *k* primeras.

Si se pasa por la isla *k* al ir de *i* a *j*, entonces la anchura es el mínimo entre la anchura máxima del camino que va de *i* a *k* (que ya no pasa por *k* como isla intermedia pues los ciclos se pueden descartar) y la anchura máxima del camino que va de *k* a *j* (que tampoco pasa por *k*).

Como el coste del camino completo se obtiene haciendo el mínimo entre los costes de los caminos parciales, para que el primero sea óptimo también ha de serlo uno de los segundos, pero en principio no sabemos cuál. Sin embargo, si uno de ellos no es óptimo se puede mejorar y esto podría mejorar el coste total; por tanto, no perdemos ningún valor óptimo si consideramos solamente soluciones óptimas para los caminos parciales (es decir, se cumple el *principio de optimalidad*).

De esta forma, tenemos en el caso general la siguiente recurrencia:

$$A^k(i,j) = \text{má}\left\{A^{k-1}(i,j), \text{mín}\{A^{k-1}(i,k), A^{k-1}(k,j)\}\right\}.$$

El caso básico (para $k = 0$) consiste en ir directamente de i a j por el puente que las une, si este existe, o no hacer nada si $i = j$

$$A^0(i, j) = anchura[i,j].$$

Obsérvese que la estructura es la misma que la del algoritmo de Floyd (Ejercicio 13.9) cambiando solo las operaciones + por mín y mín por máx. En consecuencia, podemos optimizar el espacio adicional y utilizar una sola matriz, ya que. como en casos anteriores, la fila k y la columna k no cambian al actualizar A. Para la fila k tenemos

$$A^k(k,j) = \text{máx}\{A^{k-1}(k,j), min\{A^{k-1}(k,k), A^{k-1}(k,j)\} = A^{k-1}(k,j)$$

ya que $A^{k-1}(k, k) = +\infty$. Igual ocurre para la columna k.

Utilizamos también una matriz de decisiones *camino*[*1..n*, *1..n*] para guardar información sobre un camino óptimo, como se hizo en el Ejercicio 13.9.

La implementación, con coste en $\Theta(n^3)$ en tiempo y en $\Theta(1)$ en espacio adicional, es la siguiente:

```
fun puentes(anchura[1..n, 1..n] de real∞) dev
⟨A[1..n, 1..n] de real∞, camino[1..n, 1..n] de
0..n⟩
```

```
A := anchura;
camino[1..n, 1..n] := [0];
para k = 1 hasta n hacer
  para i = 1 hasta n hacer
    para j = 1 hasta n hacer
      temp := mín(A[i, k], A[k, j]);
      si temp > A[i, j] entonces  {es mejor pasar por k}
        A[i,j] := temp;
        camino[i, j] := k;
      fsi
    fpara
  fpara
fpara
ffun
```

Los caminos más anchos se pueden construir de la misma forma que se construyeron los caminos mínimos en el Ejercicio 13.9.

13.11. Viaje canoa

Indiana Pérez quiere planificar una aventurilla por el Amazonas. A lo largo del río hay n poblados indígenas, cuyos habitantes, al observar el creciente auge del turismo rural, han ideado un sistema de alquiler de canoas. En cada poblado se puede alquilar una canoa, la cual puede devolverse en cualquier otro poblado que esté a favor de la corriente. Consultando por Internet los costes de alquileres entre poblados, Indiana ha constatado que el coste del alquiler desde un poblado i hasta otro j puede resultar mayor que el coste total de una serie de alquileres más breves. En tal caso, es más rentable devolver la primera canoa en alguna aldea k entre i y j, y seguir camino en una segunda canoa, sin cargos adicionales por cambiar de canoa.

(a) Dar un algoritmo eficiente para determinar el coste mínimo de un viaje en canoa desde todos los posibles puntos de partida i hasta todos los posibles puntos de llegada j.

(b) Modificar el algoritmo del apartado anterior para que, además, proporcione un plan de alquileres para viajar desde el primer poblado (el más cercano al nacimiento del río) hasta el último (el más cercano a la desembocadura).

---------------------------Solución-----------------------

Apartado (a)

Desde un poblado *i* solo se puede ir a poblados más abajo en el río. Supongamos que están numerados de menor a mayor en el sentido de la corriente. La información dada es el coste de un alquiler desde *i* hasta *j* para todo posible punto de partida *i* y para todo posible punto de llegada *j > i:*

alquiler[i, j] = coste de una canoa para ir directamente desde *i* hasta *j.*

Aunque se trata de un caso particular del problema de caminos mínimos, resoluble mediante el algoritmo de Floyd (Apartado (a) del Ejercicio 13.9), vamos a presentar un algoritmo más específico. Para calculare! mejor coste para ir del poblado *i* al poblado *j* planteamos una función recursiva que compruebe todas las posibilidades:

coste(i,j) = coste *mínimo* para ir desde *i* hasta *j.*

El caso básico lo tenemos cuando los poblados son consecutivos. En ese caso solo existe la posibilidad de ir directamente en una canoa; por tanto,

$$coste(i, i+1) = alqtiiler[i, i+1].$$

Cuando los poblados no son consecutivos, tenemos también la posibilidad de hacer paradas intermedias para cambiar de canoa. Tenemos que contemplar la posibilidad de pararen cualquiera de los poblados *k* entre *i y j, y* si pasamos por *k,* la parte del camino que

va de *i* a *k* y la parte del camino que va de *k* a *j* también tienen que ser óptimas porque para que la suma de costes sea óptima también han de serlo los sumandos. Por tanto, se cumple el *principio de optimalidad* y podemos plantear la siguiente recurrencia:

$$\text{coste}(i,j) = \min\left\{ alquiler\,[i,j],\ \min_{i<k<j}\{\text{coste}(i,k) + \text{coste}(k,j)\}\right\} \text{ si } i < j-1$$

La recursión está bien fundada, en el sentido de que el número de posibles paradas intermedias (j - i) siempre decrece.

Utilizaremos una matriz llamada *coste*[1..*n*, 1..n] para calcular los valores *coste(i, j),* de la cual solo necesitaremos la mitad superior por encima de la diagonal principal. Podemos rellenar la matriz por diagonales, como en el Ejercicio 13.6, para lo que necesitamos numerar las diagonales (desde *d*=1 hasta *d*=*n*-1) y los elementos dentro de la diagonal (la fila va desde el *i*=1 al *i* = *n-d, y* la columna podemos calcularla como *j* = *i*+*d).*

Obsérvese que *d* = 1 es el caso básico (la primera diagonal encima de la principal).

El algoritmo que implementa todas estas ideas es el siguiente:

```
fun canoas1(alquiler[1..n, 1..n] de real+) dev
coste[1..n, 1..n] de real+
```

```
{inicialización}
coste := alquiler;
para d=2 hasta n-1 hacer {recorrer diagonales}
  para i=1 hasta n-d hacer {recorrer elementos de
  la diagonal}
    j := i+d;
    {cálculo del mínimo}
    para k = i+1 hasta j-1 hacer
      coste[i, j] := mín(coste[i, j], coste[i, k] +
      coste[k, j]);
    fpara
  fpara
fpara
ffun
```

Nótese que el mínimo entre no parar y sí parar se calcula inicializando la matriz *coste* con *alquiler* y después minimizando (bucle **para** *k).*

Es interesante observar que a pesar de que los caminos estén orientados (no se puede retroceder) el coste del algoritmo en tiempo sigue estando en $\Theta(n^3)$ y su espacio adicional en $\Theta(1)$.

Apartado (b)-------------------------

Para obtener un plan de alquileres, necesitamos una matriz adicional *camino*[1..n, 1..n] que indique el poblado de intercambio de canoa que permite obtener un mínimo coste. Esta matriz se inicializa con ceros, que indican que de momento no hay poblado intermedio, y se irá modificando junto con *coste.* quedando el algoritmo como sigue:

```
fun canoas2(alquiler[1..n, 1..n] de real+) dev
⟨coste[1..n, 1..n] de real+, camino[1..n, 1..n] de
0..n⟩
```

```
coste := alquiler;
camino[1..n, 1..n] := [0];
para d=2 hasta n-1 hacer {recorrer diagonales}
  para i = 1 hasta n-d hacer {recorrer elementos de la diagonal}
    j := i+d;
    para k = i+1 hasta j-1 hacer
      temp := coste[i,k] + coste[k,j];
      si temp < coste[i, j] entonces
        coste[i, j] := temp;
        camino[i,j] := k;
      fsi
    fpara
  fpara
fpara
ffun
```

Su coste es el mismo que en el caso anterior.

Para obtener el plan de alquileres entre 2 poblados *i* y *j* (*i*<*j*), habrá que procesar recursivamente la información contenida en *camino*, siendo el caso básico cuando se alcanza el valor 0.

```
proc plan(e camino[1..n, 1..n] de 0..n, e i, j: 1..n)

k := camino[i,j];
si k ≠ 0 entonces
  plan(camino, i, k);
  imprimir(k);
  plan(camino, k, j);
fsi
fproc
```

Para obtener lo pedido hacemos:

imprimir(1); plan(*camino*, 1, n); imprimir(n);

13.12. Invertir

Mr. Scrooge tiene cierta cantidad de dinero *M* que quiere invertir durante n meses. Al principio de cada mes puede elegir una de entre las 3 opciones siguientes, destinando a ella todo su dinero disponible en ese momento:

1) Comprar certificados de depósito de un mes del Banco Usureros & Co., cuya comisión fija (no depende de la cantidad invertida) en el tiempo t de compra es *GCD(t)* y cuyo rendimiento porcentual al cabo del mes es *RCD(t).* Es decir, una cantidad de dinero x invertida en el tiempo t se convierte en la cantidad (x-GCD(t)) * *RCD(t)* en el tiempo t+1.
2) Comprar bonos del tesoro de Corruplandia de 6 meses. Los gastos de compra en el tiempo t son *GBT(f)* (también fijos) y el correspondiente rendimiento a los 6 meses es *RBT(G.*
3) Guardar el dinero en un calcetín y ponerlo debajo del colchón (durante un mes).

Suponiendo que Mr. Scrooge tiene predicciones fiables de *GCD, RCD. GBT* y *RBT* para los *n* meses siguientes, desarrollar un algoritmo eficiente para calcular la cantidad máxima de dinero que puede obtener haciendo las inversiones adecuadas.

--------------------------Solución---------------------

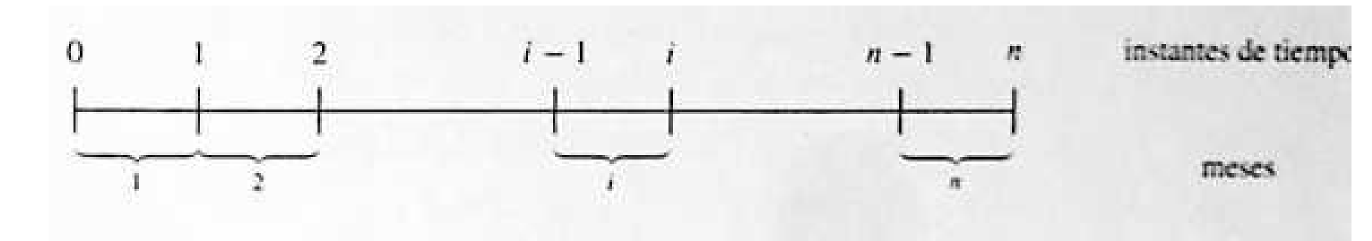


Figura 13.8: Esquema temporal para el problema de las inversiones.

Podemos invertir durante *n* meses, y para fijar los instantes en los que invertimos y en los que obtenemos los beneficios, vamos a determinar que el mes *i* empieza en el instante de tiempo i-1 y acaba en el instante de tiempo *i*, como muestra la Figura 13.8. Por tanto, el beneficio obtenido por el dinero invertido en el instante de tiempo *i*-1 (o *i*-6) se cobra en el instante *i*.

Definimos una función

inversiones(n) = *máxima* cantidad obtenible después de invertir *n* meses, es decir, en el instante n.

En el instante 0 tenemos la cantidad *M*, por lo que el caso básico es

inversiones(0) = M.

Veamos lo que podemos obtener en el instante *i* dependiendo de lo que hayamos hecho anteriormente. En primer lugar hay que observar que no conviene invertir de forma fragmentada, es decir, que si una posibilidad es la mejor conviene invertir todo el capital disponible en ella. Además, cuanto más se invierta en esa posibilidad mejor; por tanto, hay que maximizar el dinero que se obtiene en las etapas intermedias para poder invertirlo en las siguientes. Consideremos las siguientes

posibilidades:

- Si en el instante *i*-1 guardamos el dinero que tenemos, que si se ha obtenido de forma óptima será *inversiones(i*-1). en el instante *i* tendremos la misma cantidad.
- Si en el instante *i*-1 invertimos la cantidad que tenemos (que habrá sido obtenida de forma óptima) *inversiones(i*-1) en comprar certificados de depósito, al finalizar el mes *i* obtendremos un beneficio de *(inversiones(i*-1) - *GCD[i*-1]) * *RCD[i*-1].
- Si en el instante *i*-6 compramos bonos del tesoro con el dinero obtenido de forma óptima *inversiones(i*-6). al finalizar el mes *i* tendremos *(inversiones(i*-6) - *GBT[i*-6]) * *RBT[i*-6). Esta posibilidad solo se ha de considerar a partir del sexto mes.

Como nos interesa maximizar la cantidad final obtenida hacemos el máximo entre las posibilidades disponibles, quedando la recurrencia como sigue:

$$\text{inversiones}(i) = \begin{cases} \text{máx}\{\ \text{inversiones}\ (i-1), \\ (\ \text{inversiones}\ (i-1) - GCD[i-1]) * RCD[i-1]\} & \text{si } 1 \le i < 6 \\ \text{máx}\{\ \text{inversiones}\ (i-1), \\ (\ \text{inversiones}\ (i-1) - GCD[i-1]) * RCD[i-1], \\ (\ \text{inversiones}\ (i-6) - GBT[i-6]) * RBT[i-6]\} & \text{si } i \ge 6 \end{cases}$$

El algoritmo que implementa estas ideas rellenando un vector *inversiones[0..n].* con un coste en tiempo de ejecución y en espacio adicional en Θ(n), es el siguiente:

```
fun inversión (M: real+, GCD[0..n-1),
RCD[O..n-1], GBT(O..n-1]. RBT[O..n-1| de
real+) dev maxGan : real+
var inversiones[0..n]

inversiones[0]:= M;  {empezamos con una
cantidad M}
  para i=1 hasta mín(5, n) hacer
    inversiones[i] := máx(inversiones[i-
    1],(inversiones[i-1] - GCD[i-1]) * RCD[i-1]);
  fpara

  para i=6 hasta n hacer
    inversiones[i] := máx(inversiones[i-1],
    máx((inversiones[i-1] - GCD[i-1]) * RCD[i-
    1],(inversiones[i-6] - GBT[i-6]) * RBT[i-6]));
  fpara
 maxGan := inversiones[n];
ffun
```

13.13. Transformar cadenas

Sean A = $a_1 \ldots a_n$ y $B=b_1 \ldots b_m$ dos cadenas sobre un alfabeto finito de caracteres. Se desea **transformar** *A* en *B* utilizando una serie de cambios de caracteres de las 3 siguientes clases:

- **Insertar**(c, *k)* : Inserta el carácter *c* en la posición *k* de la cadena.
- **borrar**(f) : Borra el carácter en la posición *k* de la cadena.
- **Sustituir**(c, *k)* : Sustituye el carácter en la posición *k* de la cadena por el carácter c.

Por ejemplo, la cadena ***abbc*** se transforma en ***babb*** mediante los 3 siguientes cambios:

– borrar la *a* (quedando bbc),
– insertar una *a* entre las *b (babe)*
– sustituir la *c* por una *b.*

También se puede conseguir mediante solo 2 cambios: insertar *b* al principio *(**babbc**)* y borrar la c.

Desarrollar un algoritmo para saber cuál es el *número mínimo de cambios* necesarios para transformar *A* en *B* y cuáles son tales cambios.

----------------------------Solución--------------------

Aunque en principio el conjunto de posibles transformaciones para convertir A en *B* es infinito, porque se pueden considerar inserciones, borrados y sustituciones en todas las posiciones de todos los caracteres en el alfabeto dado (incluso siendo este finito), el requisito de que las transformaciones tengan un

número *mínimo* de cambios nos permite considerar solamente las transformaciones en las cuales no se realizan cambios redundantes. Por ejemplo, una transformación que realiza el borrado de un carácter recién insertado es equivalente a no hacer nada y esta segunda transformación es obviamente mejor que la primera porque tiene dos cambios menos; dos sustituciones sucesivas en una misma posición son equivalentes a una única sustitución en esa posición; una sustitución de un carácter recién insertado es equivalente a la inserción del carácter introducido por la sustitución: etc.

En particular, en tales transformaciones sin cambios redundantes no interesa considerar caracteres que no aparecen ni en *A* ni en *B.* puesto que si uno de esos caracteres se insertara o sustituyera, más adelante tendría que desaparecer con un borrado u otra sustitución. Más concretamente, los únicos cambios que se pueden realizar en una transformación sin cambios redundantes de *A* en *B* son los siguientes:

- borrar uno de los caracteres de A,
- insertar uno de los caracteres de *B.*
- sustituir un carácter de *B* en el lugar de uno de *A.*

De hecho existen todavía más restricciones sobre el número de veces que tales cambios pueden hacerse y sobre las posiciones en las cuales se

pueden hacer, pero lo que más nos interesa es hacer notar que el orden en que se realizan tales cambios (olvidando la posición en la que tiene lugar cada cambio) no tiene importancia. Por ejemplo, la transformación del enunciado

$$abbc \xrightarrow{\text{borrar } (a)} bbc \xrightarrow{\text{insertar } (a)} babc \xrightarrow{\text{sustituir } (b,c)} babb$$

es equivalente, entre otras, a

$$abbc \xrightarrow{\text{borrar } (a)} bbc \xrightarrow{\text{sustituir } (b,c)} bbb \xrightarrow{\text{insertar } (a)} babb,$$

donde (solamente para este ejemplo) hemos modificado la notación del enunciado para poner el énfasis sobre el carácter que se modifica y no en la posición en la cual tiene lugar el cambio.

Así pues, dada una transformación cualquiera entre *A* y *B* sin cambios redundantes, podemos considerar los cambios que la forman en cualquier orden. El orden que nos va a interesar para el algoritmo que desarrollamos a continuación es aquel en que los cambios se hacen desde el final de las cadenas. De esta forma, para transformar $a_1...a_n$ en $b_1...b_m$ vemos qué cambios no redundantes podemos hacer con los últimos caracteres de cada cadena a_n y b_m:

- si a_n se borra, hay que transformar $a_1...a_{n-1}$ en $b_1...b_m$ y se ha hecho un cambio.
- si b_m se inserta, queda por transformar $a_1...a_n$ en $b_1...b_{m-1}$ *y* se ha hecho un cambio.
- si $a_n = b_m$, hay que transformar $a_1...a_{n-1}$ en $b_1....b_{m-1}$, y no se ha hecho ningún

cambio.

- si $a_n \neq b_m$, se puede sustituir b_m en lugar de a_n y entonces queda por transformar $a_1...a_{n-1}$ en $b1...b_{m-1}$ habiendo hecho un cambio.

Generalizando esta idea a posiciones *i y j*, definimos la función

cambios(i, j) = número *mínimo* de cambios para transformar $a_1...a_i$ en $b_1...b_j$.

Por supuesto, el valor que nos interesa de esta función es *cambios(n, m).*

Si tomamos

$$\text{compara}(i,j) = \begin{cases} 0 & \text{si } a_i = b_j \\ 1 & \text{si } a_i \neq b_j \end{cases}$$

tenemos según la discusión anterior (y aplicando el principio de optimalidad) la recurrencia

$$\text{cambios}(i,j) = \text{mín}\begin{cases} \text{cambios}(i-1,j)+1 & \text{(borrar } a_i \text{ en la posición } j+1) \\ \text{cambios}(i,j-1)+1 & \text{(insertar } b_j \text{ en la posición } j) \\ \text{cambios}(i-1,j-1)+\text{compara}(i,j) & \text{(sustituir } a_i \text{ por } b_j \text{ en la posición } j \\ & \text{o no hacer nada si son iguales)} \end{cases}$$

para 1 < *i* < *n y* 1 < *j* < *m.*

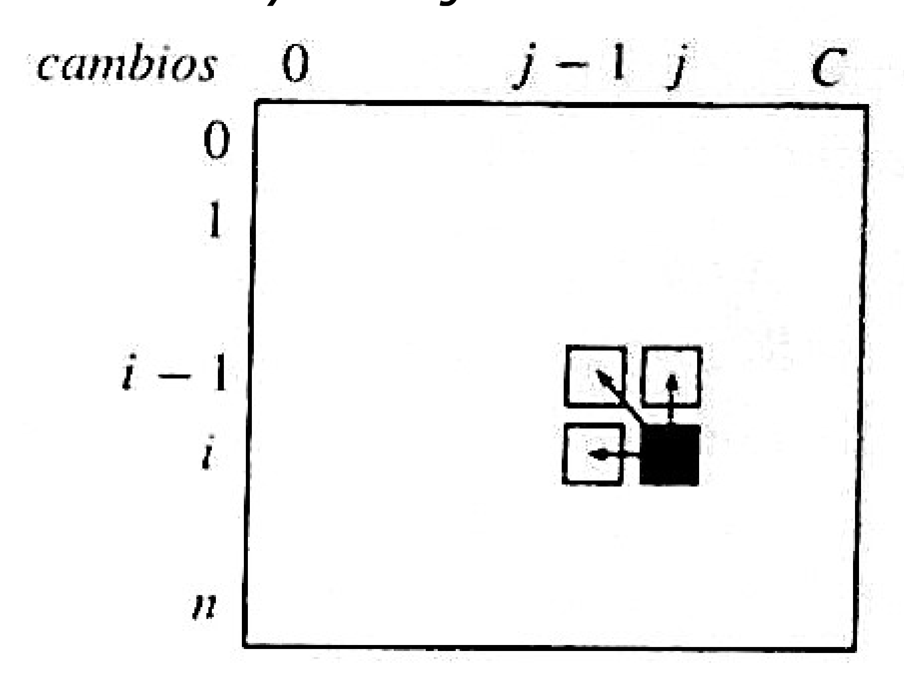


Figura 13.9. Esquema de tabla para el problema de la transformación de cadenas.

Los casos básicos se presentan cuando

ya hemos conseguido la cadena de llegada pero sobran caracteres en la de partida.

cambios(i, 0) = *i.*

que corresponde a borrar los *i* caracteres sobrantes (pues en esta situación inserciones y sustituciones no tienen sentido), o cuando ya no quedan caracteres en la cadena de partida pero faltan en la de llegada.

cambios(0, j) = j.

que corresponde a insertar los *j* caracteres necesarios (pues en esta situación borrados y sustituciones no tienen sentido).

Aplicando el método de programación dinámica, consideramos una matriz *cambios[0..n,* 0..m], Dejando aparte los casos iniciales, para calcular cada entrada (i,j) en la tabla se necesitan las entradas *(i* - I, *j* - l).(t- 1. y) de la fila anterior y la entrada (í. *j* - 1) de la misma fila pero de la columna anterior, como se ilustra en la Figura 13.9. Por eso vamos a rellenar la matriz por rilas de arriba abajo y. dentro de cada fila, de izquierda a derecha.

Como en todos los casos se trata de un mínimo entre 3 posibilidades, a partir de toda la matriz se pueden calcular los cambios necesarios partiendo de *cambios[n,m],* que es el número buscado. Si *cambios[n,* m] = *cambios[n*-1, m] + 1, se ha borrado el carácter a_n en la posición *m*+1; si es

igual a *cambios[n, m*-1] + 1, se ha insertado b_m en la posición *m*; etc. Nótese que aunque una de las posiciones consultadas está en la misma fila, no podemos optimizar el coste en espacio, ya que es necesario consultar *2* posiciones en la fila anterior.

La implementación es la siguiente, donde *numCambios* es el número mínimo de cambios necesarios para transformar A[1..n] en B[1..m] y el vector *quéCambios* indica qué transformaciones hay que hacer. Como no sabemos el número de cambios a priori, *quéCambios* se declara de tamaño *n* + *m* que es una cota superior de tal número; la parte interesante del resultado estará en *quéCambios[*1..*númCambios].* El algoritmo tiene un coste en Θ(nm) tanto en tiempo como en espacio.

tipos
    operación = {borrar, insertar, sustituir}
    cambio **= reg**
        op: operación
        pos: nat
        car: car
    freg
ftipos

fun transformar(A[1..n], B[1..m] **de** *car*) **dev** ⟨*númCambios: nat*, *quéCambios[1..n+m*] **de** *cambio*⟩
var *cambios[0..n, 0..m*] **de** *nat*

{inicialización}
para *i* = 0 **hasta** *n* **hacer**

```
cambios[i,0] := i;
fpara

para j = 1 hasta m hacer
cambios[0,j] := j;
fpara

{rellenamos la matriz}
para i=1 hasta n hacer
 para j=1 hasta m hacer
  si A[i] = B[j] entonces
  compara := 0;
  si no compara := 1;
  fsi
  cambios[i, j] := mín(mín(cambios[i-1, j] + 1,
  cambios[i, j-1] + 1), cambios[i-1, j-1] +
  compara);
 fpara
fpara
númCambios:= cambios[n, m];

{cálculo de las transformaciones}
k := númCambios;
i := n;
j := m;
mientras k > 0 hacer
 casos
  cambios[i, j] = cambios[i-1,j] + 1 →
 quéCambios[k].op := borrar;
 quéCambios[k}.pos := j+1;
 k := k-1;
 i := i-1;

 ▯ cambios[i, j] = cambios[i, j-1] + 1 →
 quéCambios[k].op := insertar;
 quéCambios[k].pos := j;
 quéCambios[k].car := B[j];
 k := k-1;
 j := j-1;

 ▯ cambios[i, j] = cambios[i-1, j-1] + 1 →
 quéCambios[k].op := sustituir;
 quéCambios[k].pos := j;
 quéCambios[k].car := B[j];
 k := k-1;
 j := j-1;
 i := i-1;
```

```
 cambios[i, j] = cambios[i-1, j-1] →
j := j–1;
i := i-1;
fcasos
fmientras
ffun
```

Se puede demostrar que cuando $a_n = b_m$ las transformaciones óptimas de $a_1 \ldots a_n$ en $b_1 \ldots b_m$ coinciden con las transformaciones óptimas de $a_1 \ldots a_{n-1}$ en $b_1 \ldots b_{m-1}$. Esto da lugar a una recurrencia para la función *cambios* que es ligeramente diferente de la anterior, porque cuando *ai = bj* no se calcula un mínimo entre 3 posibilidades, sino que se considera directamente una de ellas.

$$\text{cambios}(i,j) = \begin{cases} \text{cambios}(i-1, j-1) & \text{si } a_i = b_j \\ min \left\{ \begin{matrix} \text{cambios}(i-1, j) + 1 \\ \text{cambios}(i, j-1) + 1 \\ \text{cambios}(i-1, j-1) + 1 \end{matrix} \right\} & \text{si } a_i \neq b_j \end{cases}$$

para $1 \leq i \leq n$ y $1 \leq j \leq m$. Los casos básicos no cambian:

cambios(i, 0) **=** *i*

cambios(0, j) = *j.*

El algoritmo correspondiente a esta versión se implementa de forma parecida a la anterior.

13.14. Programas en cintas

Tenemos que almacenar n programas en 2 cintas, cada una de longitud *L.* siendo *li* la longitud de cinta necesaria para almacenar el programa *i.* Suponemos que $\sum_{i=1}^{n} l_i \leq L$. Un programa puede ser almacenado en cualquiera de las 2 cintas. Si S_1 es el conjunto de programas en la cinta 1, el tiempo de acceso en el peor caso a un programa cualquiera es proporcional a

$$\text{máx}\left\{\sum_{i\in S_1} l_i, \sum_{i\notin S_1} l_i\right\}$$

Una asignación óptima de programas a cintas minimiza el tiempo de acceso en el peor caso. Desarrollar un algoritmo para determinar el tiempo de acceso en el peor caso de una asignación óptima y dicha asignación.

---------------------------Solución----------------------

En todo lo que sigue $L' = \sum_{i=1}^{n} l_i$.

Primera solución (ineficiente)

Primero vamos a presentar las ideas de una solución poco eficiente en espacio que luego mejoraremos, pero que ayuda a comprender las siguientes.

Definimos una función

tiempo(i, L_1, L_2) = tiempo de acceso óptimo cuando faltan por asignar los programas del 1 al *i*, la ocupación de la primera cinta es L_1 y la de la segunda es L_2.

El caso recursivo, aplicando el principio de optimalidad. se obtiene al considerar

las 2 posibilidades para asignar el programa *i:*

$$tiempo(i, L_1, L_2) = \text{mín}\{\underbrace{tiempo\ (i-1, L_1 + l_i, L_2)}_{\substack{\text{programa } i \\ \text{en cinta 1}}}, \underbrace{tiempo\ i-1, L_1, L_2 + l_i)}_{\substack{\text{programa } i \\ \text{en cinta 2}}}\}$$

con $1 \leq i \leq n$, $0 \leq L_1$, $L_2 \leq L'$, aunque los límites de L_1 y L_2 habría que refinarlos en la implementación, para no salimos de la matriz donde se van guardando valores (ver siguiente solución).

El caso básico se presenta cuando no quedan programas por asignar. En ese caso, según el enunciado,

$$tiempo(0, L_1, L_2) = \text{máx}\{L_1, L_2\} \quad 0 \leq L_1, L_2 \leq L'.$$

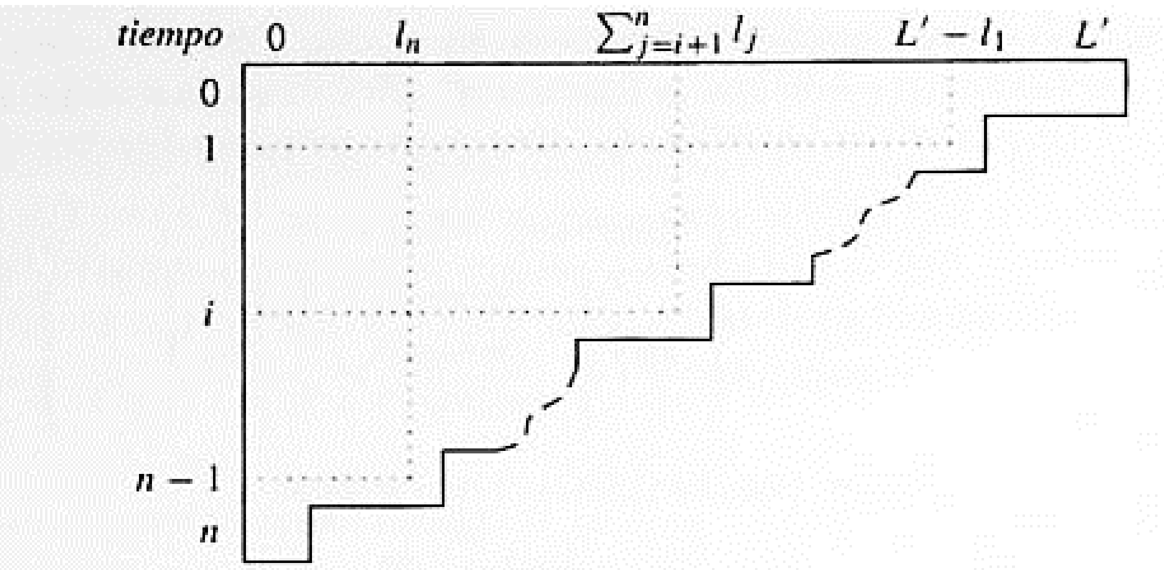


Figura 13.10: Esquema de tabla para el problema de programas en cintas.

La solución viene dada por *tiempo(n,* 0, 0) y el coste en tiempo y espacio de un algoritmo que implemente estas ideas sería $0(nL'^2)$. Pero es fácil mejorar esos costes, como muestra la siguiente solución.

Segunda solución (más eficiente)

Definimos *tiempo(i, L_1)* como antes pero nos olvidamos de la ocupación de la segunda cinta, porque en ella están todos los que faltan en la primera, y podemos calcular inmediatamente su ocupación porque un programa ocupa lo mismo en una cinta que en otra.

El caso recursivo es

$$tiempo\ (i, L_1) = min\{\underbrace{\text{tiempo}\ (i-1, L_1 + l_i)}_{\text{programa } i \text{ en cinta 1}}, \underbrace{tiempo(\ i-1, L_1)}_{\text{programa } i \text{ en cinta 2}}\}\ 1 \leq i \leq n, 0 \leq L_1$$

La recursión está bien fundada porque el primer argumento decrece.

El caso básico es

tiempo(0, L_1) = máx{L_1, L'- L_1} 0≤ L_1≤ *L'.*

La solución viene dada por *tiempo(n,* 0).

Utilizamos una matriz *tiempo[0..n. 0..L']*. Como en las llamadas recursivas el segundo argumento puede crecer, para no realizar accesos fuera de la matriz hay que ser cuidadosos con el límite de L_1. Cuando estamos considerando el programa *i* ya hemos grabado los programas del *i*+1 al *n.* Es decir, cuando rellenamos la fila i, las columnas que tenemos que rellenar son, como mucho, hasta $\sum_{j=i+1}^{n} l_j$ (*representando el caso en el que todos los programas se hayan grabado en la primera cinta*). La Figura 13.10 muestra la parte de la matriz que hay que rellenar.

El algoritmo, cuyo coste en tiempo y

espacio adicional está ahora en Θ(nL'), es el siguiente:

$$\left\{L' = \sum_{i=1}^{n} longitud\,[i]\right\}$$

```
fun dosCintas1(longitud[1..n] de nat⁺, L': nat⁺)
dev ⟨tiempoOptimo: nat, asignación[1..n] de
1..2⟩
{el programa i se almacena en asignación[i]}
var tiempo[0..n, 0..L'] de nat

{inicialización}
para j = 0 hasta L' hacer
   tiempo[0, j] := máx(j, L'-j);
fpara
{rellenar la matriz}
suma:= L';
para i = 1 hasta n hacer
   suma := suma - longitud[i]; {suma = Σ_{j=i+1}^{n} longitud [i]}
   para j = 0 hasta suma hacer
      tiempo[i, j]:=mín(tiempo[i-1, j+longitud[i]],
      tiempo[i–1, j]);
   fpara
fpara
tiempoOptimo:= tiempo[n, 0];

{cálculo de la solución}
L1:= 0
para i = n hasta 1 paso - 1 hacer
   si tiempo[i, L1] = tiempo[i-1, L1] entonces
      asignación[i]:= 2; {colocamos el programa i en la
      cinta 2}
   si no
      asignación[i]:= 1; {colocamos el programa i en la
      cinta 1}
      L1:= L1 + longitud[i];
   fsi
fpara
ffun
```

Tercera solución (con límites más sencillos)

Podemos pensar una tercera solución, que simplifica el recorrido de la matriz, al rellenarla completamente. Esta solución es similar a la anterior, pero en vez de llevar como segundo argumento el espacio ocupado en la primera cinta, vamos a llevar como argumento el espacio libre.

Definiendo

tiempo(i, L_1) = tiempo de acceso óptimo cuando faltan por asignar los programas del 1 al i y el espacio libre en la primera cinta es L_1.

la solución viene dada por *tiempo(n, L).*
El caso recursivo es

$$\text{tiempo}(i, L_1) = \begin{cases} \text{tiempo}(i-1, L_1) & \text{si } l_i > L_1 \\ min\{tiempo(i-1, L_1 - l_i), \text{tiempo}(i-1, L_1)\} & \text{si } l_i \leq L_1 \end{cases}$$

con 1 < *L1* < *n.* 0 < *L* < *L.* El caso básico es

tiempo(0, L_1) = máx{L – L_1, *L' - (L -* L_1)}

con $0 \leq L_1 \leq L$. Nótese que esta fórmula se ha complicado porque es necesario calcularla en términos de espacio ocupado en vez del espacio libre que lleva el segundo argumento.

Utilizamos una matriz *tiempo[0..n,* 0..L], que rellenamos completamente, recorriéndola por filas de arriba abajo, y por columnas de izquierda a derecha. El algoritmo es el siguiente:

(L' = £"=1 longitiid[i] }

```
fun dosCintas2(longitud[1..n] de nat+. L, L':
naf')
dev {tiempo-óptimo : nat. asignación[l..ii] de
1..2)
( el programa i se almacena en asignación[i ] }
var tiempo[0..n. 0..L] de nat
{ inicialización )
para j = 0 hasta L hacer
tiempo[0, j] := máx(¿ — j. L' — (L — j))
fpara
{ rellenamos la matriz )
para i = 1 hasta n hacer
para j = 0 hasta L hacer
si longitud[i] > j entonces
tiempo[i, j] := tiempo[i — 1. j]
si no
tieinpo[i. j] := mín(íiempo[í - 1. j - longitnd[i]],
tiempo[i — 1, j])
fsi
fpara
fpara
tiempo-óptimo := tiempo[n,
{ cálculo de la solución )
¿1 := L
para i = n hasta 1 paso - 1 hacer
si tiempo[i, ¿i] = tiempo[i — 1, ¿i] entonces
asignación[i] := 2  { colocamos el programa i en
la cinta 2}
si no
asignación[i] := 1  {colocamos el programa i en
la cinta 1}
L1 := L1 - longitud[i];
fsi
fpara
ffun
```

El coste en tiempo y espacio adicional está en Θ(nL), pero se podría reducir a O(nL') utilizando *L'* como límite en la matriz auxiliar porque sabemos que $L' \leq L$.

13.15. Procesadores

Una serie de *n* tareas ha de ser procesada en un sistema que cuenta con dos procesadores *A y B.* Para cada tarea *i* se conocen los tiempos *ai, bi* que cada uno de los procesadores necesita para realizarla. Debido a las características de los procesadores y de las tareas es posible que para una tarea *i* se tenga *ai > bi* mientras que para otra tarea *j / i* sea *ai < bj.* Nótese que una tarea no puede dividirse entre los procesadores. Obtener un procedimiento para asignar las tareas a los procesadores de forma que se minimice el tiempo necesario para terminar todas ellas.

----------------------------Solución--------------------

Primera solución (ineficiente)

Definimos una función
tiempofj, T_A, T_B*)* = tiempo mínimo para procesar las *i* primeras tareas.
cuando el procesador *A* ya está ocupado un tiempo *T,\ y* el procesador *B* un tiempo *T/i.*

El valor buscado, por tanto, es *tiempo(n,* 0. 0). Cuando asignamos la tarea *i* podemos asignarla a uno u otro procesador, y quedarnos con la mejor opción. El resto del reparto se tiene que hacer de forma óptima, es decir, se cumple el principio de optimalidad, por lo que
tiempoíj, T_A, T_B*) = mín{tiempo(i -* 1, T_A + *ai.* T_B*). tiempoij* — 1. T_A. T_B 4- ¿,)1,
con 1 < *i* < *n,* 0 < T_A < "^*t*=\ ª*t* Y 0 — ?a

— ZL/=| *bl* (aunque los límites de T_A y T_B se refinarán en la implementación).

El caso básico se presenta cuando ya no quedan tareas que asignar, en cuyo caso el tiempo necesario para terminar todas las tareas será el máximo de los tiempos de proceso en cada procesador,

tiempo(fi,T_A,T_B) = máxtT^, Tg).

En este caso no podemos optimizar la función reduciendo los tres argumentos a dos como se hace en la segunda solución del Ejercicio 13.14. ya que saber el tiempo que tardan en ejecutarse las tareas asignadas a un procesador no ayuda a saber el tiempo que tardan las tareas asignadas al otro procesador. Por tanto, utilizaremos una matriz tridimensional *tiempo[0..n,* O..S,|, O..5g| donde S i = E"=i *"i y* ^{S}*B* = E"=i

Como en la segunda solución del Ejercicio 13.14, para no hacer accesos fuera de la matriz, hay que ser cuidadosos con los rangos de los índices. Cuando estamos considerando asignar la tarea *i.* ya hemos asignado las tareas de la i + 1 a la *n.* por lo que solo tienen sentido las casillas *tiempo[i. j. k*) tales que 0 < y < Z7=í+i ªi Y 0 - *k* - *Xt=i*₊*\ bl-* Rellenando solo estas posiciones nunca accederemos a posiciones fuera de la matriz.

El algoritmo es el siguiente:

tipos

procesador = (proc-a. proc-b)

hipos

i 'Vt = EL, /H/] A s_B = E;'=I J5[/I }

fun dos-procesadores1 (A[I.JI], B[I..n| **de** *nat~. S_A. S_B : naf~)*
dev (*tiempo-óptimo : nat. asigiiación[* **I.** *n* **| de** *procesador)*

(el programa *i* se procesará en *asigmtción[i] }*

```
var tiempo[p..n. O..SA. O..Sg] de nat
  ( inieialización )
  para / = 0 hasta 5a hacer
    para k = 0 hasta SB hacer
      tiempo[0. j. A] := máx(j. k)
    fpara
  fpara
  ( rellenar la matriz tridimensional )
  sunut-A := S..| : stima-B := SB
  para i = I hasta n hacer
    sttma-A := suina-A — zA[z]    {
suma-A = E"=í+ı )
    snma-B := sama-B - B[/]   ( snma-B
= ^2"=i+i fi[/| )
    para 7=0 hasta suma-A hacer
      para k = 0 hasta suma-B hacer
        tiempo[i. j.k] := mín(tiempo[i - \.
        j + A[i], k], tiempo[i - 1. j. k +
        B[f]|)
      fpara
    fpara
  fpara
  tiempo-óptimo := tiempo[n. 0. 0]
  1 cálculo de la solución )
  TA := 0 : Tu := 0
  para i — n hasta I paso - I hacer
    si tiempo[i. T.\. TB] = tiempo[i - I,
      T\ + A[¿]. TB] entonces
      asigmicióti[i | := proc-a { la tarea i
      se ejecuta en A } TA-.= TA +A(/|
    si no
      asignación[i ] = proc-b ( la tarea i
      se ejecuta en B )
      TB = TB+B{i\
    fsi
```

fpara
ffun

El coste de este algoritmo está en &(nS_AS_B) tanto en tiempo (por los tres bucles) como en espacio adicional (por la matriz tridimensional).

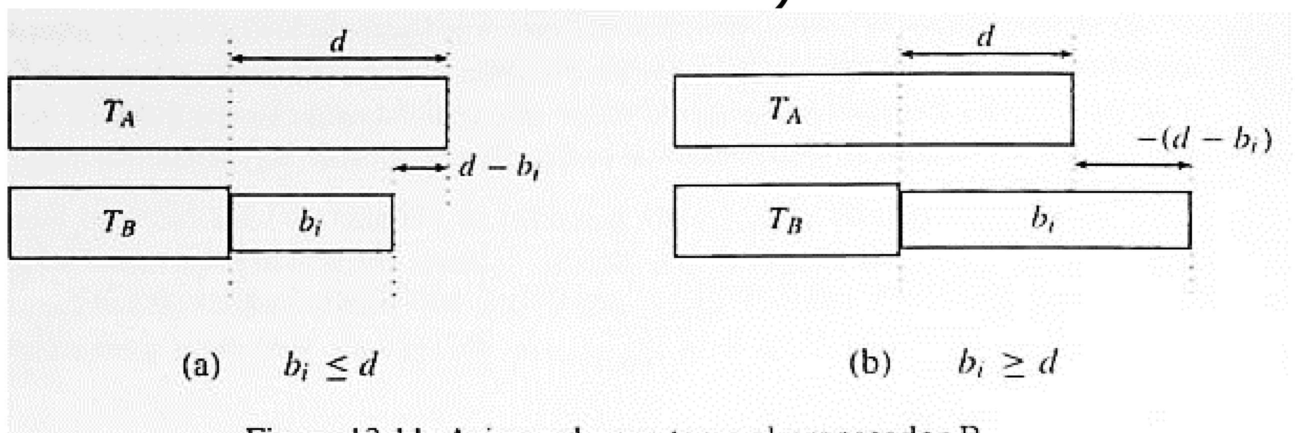


Figura 13.11: Asignando una tarea al procesador B.

Segunda solución (más eficiente)

Como hemos comentado antes, no podemos optimizar el tiempo y espacio adicional del algoritmo guardando solo información de uno de los procesadores, análogamente a como se hace en la segunda solución del Ejercicio 13.14, sino que tenemos que pensaren otra idea más sofisticada. Lo que guardaremos será la diferencia entre el tiempo ocupado en el procesador A y el tiempo ocupado en el procesador *B.* Nótese que esta diferencia puede ser positiva si *A* va por delante, es decir. 4 tarda más que *B* en ejecutar las tareas que tiene asignadas; o negativa, si *B* va por delante.

La función que definimos es la siguiente:

tiempo(i. d) = tiempo mínimo

para procesar las *i* primeras tareas.

partiendo del tiempo inicial (<7, 0) si $d > 0$.

o del tiempo inicial (0. — *d)* si $d < 0$.

Hay que notar que solo guardamos como argumento la diferencia entre los tiempos de los procesadores, y no el tiempo total que tardan las tareas en ejecutarse en los procesadores. Esto habrá que tenerlo en cuenta a la hora de hacer las llamadas recursivas. El tiempo en el que ambos procesadores están ocupados (tiempo acumulado en los dos procesadores por igual) tendrá que ir contándose fuera de las llamadas recursivas.

El valor buscado es *tiempo(n,* 0), el tiempo óptimo que tardan en ejecutarse todas las tareas cuando los dos procesadores están inicialmente libres, y por tanto la diferencia de tiempos es 0.

Veamos qué posibilidades hay al asignar la tarea *i.* Distinguimos casos según el signo de *d:*

- si $d > 0$ (el procesador *A* va por delante) podemos
  - asignar la tarea *i* al procesador A, por lo que la diferencia de tiempos aumenta y no cambia el tiempo acumulado (porque sigue siendo igual en ambos procesadores); o
  - asignar la tarea *i* al procesador *B,*

por lo que la diferencia disminuye (pudiendo llegar a ser negativa), y el tiempo acumulado tiene que incrementarse. El esquema de la Figura 13.11 muestra cuánto tiene que incrementarse cuando ó, < *d* y cuando *b, > d.* En ambos casos, el tiempo acumulado se incrementa en la cantidad mín(r/, *bi}.*

Por tanto, en este caso, y ya que el resto de las tareas tiene que asignarse de forma óptima (se cumple el principio de optimalidad), la función se define de la siguiente manera:

lienipo(J,d) = mín{tiempo(i — \.d + ai), liempo(i — l.d — bi) + mín{</, í>,})

- si *d* < 0 (el procesador *B* va por delante) la distinción de casos es análoga y simétrica a la anterior, pero ahora hay que tener cuidado con el signo de *d.* La función se define como:

tiempo(i, d) = mm\tiempo(i — 1, *d* + a,) + mín(—*d,* a,), *tiempolj* — 1, *d* — ó,))

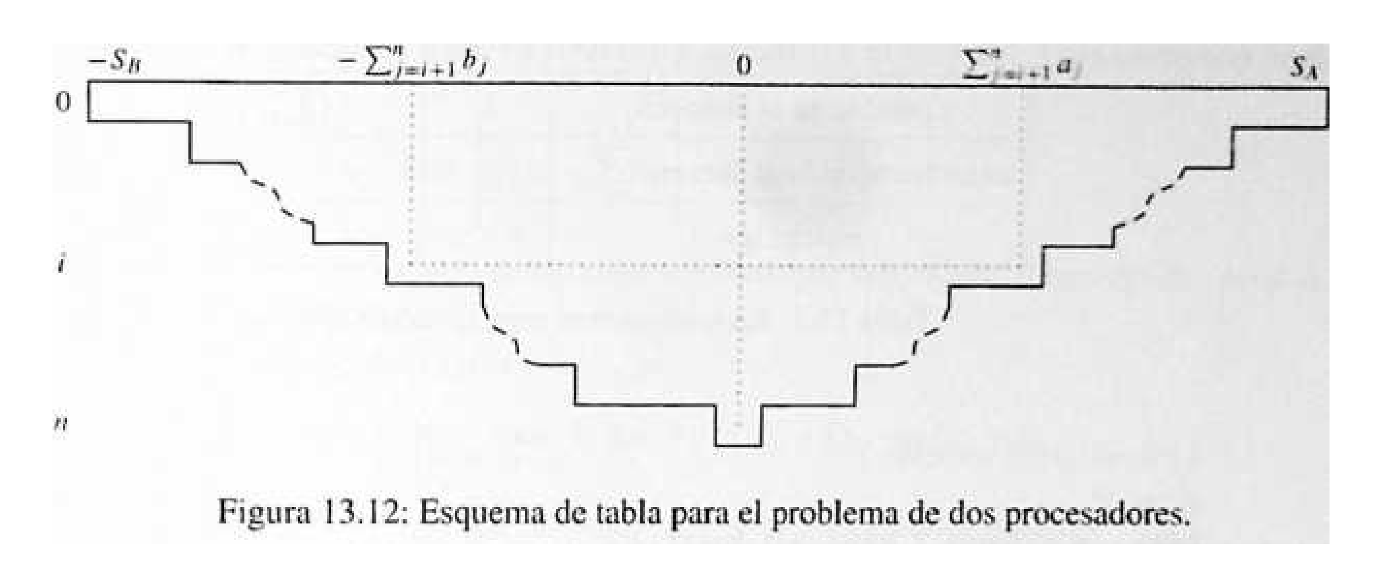


Figura 13.12: Esquema de tabla para el problema de dos procesadores.

El caso básico se presenta cuando ya no hay más tareas que asignar. En ese caso, definimos
tiempo{0, d) = | d |,
dado que si no hay tareas que asignar, el tiempo consumido es el máximo entre los tiempos de los dos procesadores, pero como la situación ahora es (<7. 0) o (0. *—d)* (dependiendo del signo de </). el máximo es el valor absoluto de *d.*

Utilizamos una matriz *tiempo[0..n, —*
| donde S-i = £"=1«/ y *SB* = £J=I ^/- Mue rellenare
mos por filas de arriba abajo y por columnas de izquierda a derecha. Igual que en el apartado anterior, hay que ser cuidadosos con los índices, para no hacer accesos fuera de la matriz. La Figura 13.12 muestra la parte de la matriz que hay que rellenar.

Si queremos utilizar la tabla para recuperar la asignación de tareas, no podemos reducir el espacio adicional a un vector. El algoritmo, de coste en 0(//(5a + Sa)) tanto en tiempo como en espacio adicional, es el siguiente:

{ S.A = E;'=| -4[/J A *SB* = H=1 5[/J 1

fun dos-procesadores2G4[l..ti], B[l../t] **de** *nat**. 5a- *$B* ■ *naF)*
dev *{tiempo-óptimo : nat. osignación[l..n] de procesador)* j el programa *i* se procesará en *asignación[i]* *)* **var** *tiempo[0..n, —SB-S,\]* **de** *nat*
{ inicialización }
para *j* = *-Su* **hasta** *S,* **hacer**
tiempo[0,j] := |j|
fpara
{ rellenar la matriz }
suma-A := S,\ : suma-B := —Su
para *i* — **I hasta n hacer**
suma-A := suma-A — A[í]*{ suma-A =* 52"=IJ-I •’WI I
suma-B := suma-B + B[í] *{ suma-B* = — *Yj=i+i* **1**
para *j* = *suma-B* **hasta** - 1 **paso** - 1 **hacer**
tiempo[i, j] := min(tiempo[i - **1,** *j* **+ A[í]] +** *mín(-j.* **A[í]).** *tiempo[i* **- 1.** *j* -
fpara
para *j* = **0 hasta** *suma-A* **hacer**
tiempofi, j] := mín(tiempo[i — **1.** *j* **- M(iJ).** *tiempo[i* **- 1.** *j - B[i***|| + min(y.** *B[i* **11) fpara**
fpara
tiempo-óptimo := tiempo[n, **0]**

	A_1	A_2
cantidad en el almacén	x	$(\sum_{l=1}^{k} c_l) - x$
cantidad servida al comercio C_k	j	$c_k - j$
restricción	$j \leq x$	$c_k - j \leq (\sum_{l=1}^{k} c_l) - x$

Tabla 13.1: Restricciones al repartir desde dos almacenes.

(cálculo de la solución }
d := 0
para *i* — *n* **hasta** 1 **paso** — 1 **hacer**
si *d* > 0 **entonces**
si *tiempo\i. d]* = *tiempo[i* — 1. *d* + A[/]] **entonces** *asignación[i] :=* proc-a ; *d := d* + A[z]
si no
asignación[i] := proc-b; *d := d* — *B[i]*
fsi
si no { *d* < 0)
si *tiempo[i.d]* = *tiempo\i* — *\.d* — B[/J] **entonces** *asignación[i]* := proc-b; *d := d* — B[t]
si no
asignación[i] := proc-a ; *d := d* + A[z]
fsi
fsi
fpara
ffun

13.16. Almacenes

Se tienen 2 almacenes A1 y A2 en los que se dispone respectivamente de a_1 *y* a_2 unidades (indivisibles) de cierto producto. Desde estos 2 almacenes se ha de abastecer a *n* comercios $C_1 \ldots C_n$ en cada uno de los cuales se precisa una determinada cantidad c_j (donde *j* entre 1 y *n)* de unidades del producto, de forma que $\sum_{j=1}^{n} c_j = a_1 + a_2$.

El coste del transporte desde A_i hasta C_j viene dado por una función $t_{ij}(x)$ donde x es el número de unidades del producto transportadas.

Se desea determinar las cantidades de unidades x_{ij} que cada almacén debe enviar a cada comercio de forma que el coste total del transporte $\sum_{ij} t_{ij}(x_{ij})$ sea mínimo.

--------------------------Solución-------------------------

Dado que desde el almacén A_2 se distribuye siempre lo que falta con respecto al A_1, es claro que basta con llevar la información de A_1. Así. definimos una función

coste(k, x) = coste mínimo cuando el almacén A1 tiene x unidades disponibles y A2 tiene $\left(\sum_{j=1}^{k} c_j\right) - x$ para abastecer a los comercios del C_1 al C_k.

Por tanto, el valor que queremos calcular es *coste(n,* a_1).

Cuando vamos a abastecer al comercio Ck, tenemos que decidir cuánto le sirve cada almacén. Ya que no sabemos si las funciones de coste tij son aditivas, es

decir, en general tij(x) + tij(y) ≠ *tij(x*+y), hay que considerar servir varias unidades cada vez. Si *j* es la cantidad que le sirve el almacén A1, ck-*j* será la que tiene que servir A2. Se tiene que cumplir que 0≤*j*≤ ck, y además se tienen que cumplir las restricciones de la Tabla 13.1.

Juntando las restricciones sobre la cantidad *j* servida por el almacén A1 tenemos que se tiene que cumplir

$$\text{máx}\left\{0, x - \sum_{l=1}^{k-1} c_l\right\} \leq j \leq min\{c_k, x\}.$$

Después de servir a Ck, al resto de los almacenes hay que servirlos de forma óptima, pues se cumple el principio de optimalidad.

Por tanto, la definición de la función *coste* es:

$$\text{coste}(k, x) = \min_{m \leq j \leq m'} \{\text{coste}(k-1, x-j) + t_{1k}(j) + t_{2k}(c_k - j)\}$$

donde *m* = máx{0, x — c/) *y m'* = mín{cj-, x|. Y el caso básico es *coste(0,* x) = 0.

No es necesario distinguir el otro posible caso básico,

$$\text{coste}(k, 0) = \sum_{j=1}^{k} t_{2j}(c_j)$$

ya que se puede calcular aplicando la fórmula general, si tomamos tij(0)=0.

Utilizamos una matriz coste[0..n, 0..a_1] para ir calculando los valores. La matriz se rellena por filas de arriba abajo y, cada

fila, de izquierda a derecha. Utilizamos además una matriz de decisiones *cantidad-*A_1 $[1..n, 0..a_1]$ para almacenar la cantidad servida por el almacén A1 a cada comercio. El algoritmo. que también calcula el reparto y cuyo coste en tiempo está en $\Theta(na_1^2)$ y en espacio adicional en $\Theta(na_1)$, es el siguiente:

```
fun reparto(C[1..n] de nat⁺, T₁[1 ..n, 0..a₁],
T₂[1..n, 0..a₂] de real, a₁, a₂: nat⁺)
dev ⟨costeOptimo: real, solucion[1..2, 1..n]
de nat⟩
var coste[0..n, 0..a₁] de real∞, cantidadA₁
[1..n, 0..a₁] de nat

{inicialización}
coste[0, 0..a₁]:= [0];
suma:= 0;
  {rellenamos la matriz}
  para k = 1 hasta n hacer
  { suma = Σ_{l=1}^{k-1} C[l] }
      para x = 0 hasta a₁ hacer
       {cálculo del mínimo}
       coste[k,x] := +∞;
        para j = máx(0, x - suma) hasta
        mín(C[k], x) hacer
          temp := coste[k-1, x-j] + T₁[k, j] +
          T₂[k, C[k]-j];
          si temp < coste[k,x] entonces
             coste[k,x] := temp;
             cantidadA₁[k, x] := j;
          fsi
        fpara
      fpara
    suma := suma + C[k];
  fpara
costeOptimo:= coste[n,a₁];
{recuperar la solución}
```

```
x:= a1
para k=n hasta 1 paso - 1 hacer
    solución[1,k] := cantidadA1[k, x];
    solución[2,k] := C[k] – solución[1,k];
    x:= x – solución[1, k];
fpara
ffun
```

13.17. Vacas

El Tío Antonio tiene en su granja 2 vacas: *Devoradora* y *Listilla.* Antes de ordeñarlas, las alimenta llenando una hilera de *n* cubos con pienso (*n* es par). Cada cubo *i* (para *i* entre 1 y n) contiene una cantidad *pi* de pienso indicada en el cubo (todas las cantidades son distintas). Cada vaca en su turno debe elegir el cubo de uno de los extremos y comerse su contenido. El cubo se retira y el turno pasa a la otra vaca. Se sigue así hasta agotar los cubos. La vaca que comienza comiendo se determina a priori por un procedimiento cualquiera. El objetivo de ambas vacas es comer en total lo máximo posible. La estrategia de *Devoradora* consiste en pensar poco y escoger el cubo de los extremos que esté más lleno. En cambio, *Listilla* prefiere pensárselo un poco más, para lo que ha adquirido lo último en computadoras vacunas portátiles.

(a) Demostrar que la estrategia de *Devoradora* no es óptima, incluso cuando le toca escoger primero.

(b) *Listilla* ha cursado un master en Informática por la Escuela Superior Bovina, pero en dicha escuela no estudian la programación dinámica, por lo que pide ayuda para diseñar un algoritmo utilizando dicha técnica, suponiendo que le toca empezar a escoger.

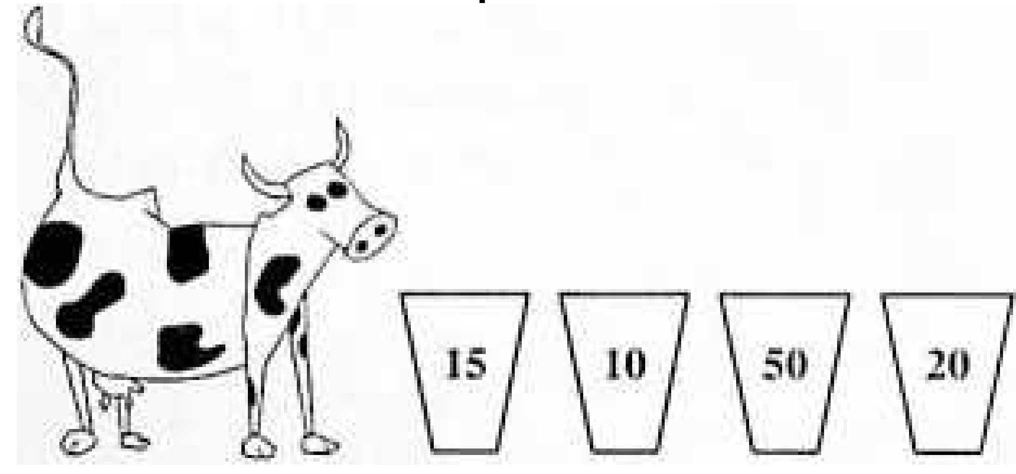
(c) Diseñar un algoritmo de coste *lineal*

de forma que *Listilla,* siempre que comience comiendo ella, coma al menos tanto como *Devoradora,* independientemente de la estrategia que siga esta última. ¿Es óptima la nueva estrategia?

-------------------------Solución-------------------------

Apartado (a)

Basta con dar un contraejemplo para demostrar que la estrategia de *Devoradora* no es óptima:



En este caso. *Devoradora* comería el cubo con 20, después *Listilla* podría comer el cubo con 50, tras lo que *Devoradora* comería el de 15 y *Listilla* el de 10. *Listilla* ha comido 60 y *Devoradora* solo 35.

Apartado (b)-----------------------

Definirnos una función

vacas(i, j) = cantidad *máxima* que come *Listilla* con los cubos desde el p_i hasta el p_j cuando le toca empezar a comer.

Lo que buscamos es *vacas*(1, *n*).

Cuando los cubos son $p_i, p_{i+1},\dots,p_j$, *Listilla* puede elegir entre comer p_i o p_j. Si elige comer p_i (respectivamente, p_j) entonces *Devoradora* comerá p_{i+1} o p_j

(respectivamente, p_i o p_{j-1}), dependiendo de cuál sea el mayor. Después le volverá a tocar a *Listilla,* que deberá seguir comiendo de forma óptima, es decir, se cumple el *principio de optimalidad*.

La definición de la función es

$$vacas(i,j) = max\left\{p_i + \begin{cases} vacas(i+1, j-1) & \text{si } p_j > p_{i+1} \\ vacas(i+2, j) & \text{si } p_{i+1} > p_j \end{cases},\ p_j + \begin{cases} vacas(i+1, j-1) & \text{si } p_i > p_{j-1} \\ vacas(i, j-2) & \text{si } p_{j-1} > p_i \end{cases}\right\}$$

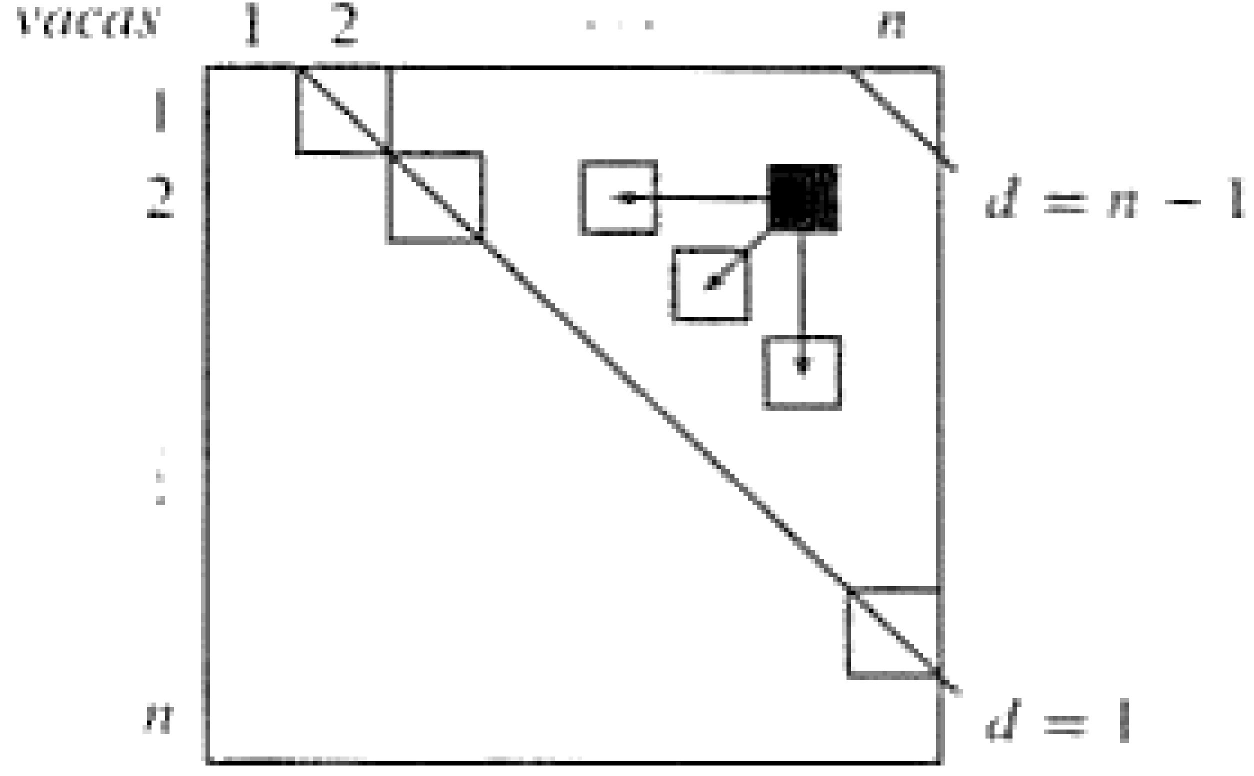


Figura 13.13: Esquema de tabla para el problema de las vacas.

cuando hay al menos 4 cubos, $j-i+1 \geq 4$, y el número de cubos es par. Cuando solo hay 2 cubos, *Listilla* coge el cubo con más comida:

$$vacas(i, i+1) = \text{máx}\{p_i, p_{i+1}\}.$$

Utilizamos una matriz *vacas*[1 ..n,. 1..n] para ir calculando los valores. Para cada posición, necesitamos 3 posiciones de la diagonal 2 posiciones más abajo, como muestra la Figura 13.13.

Vamos a recorrer la matriz por diagonales (de forma parecida a la del

Ejercicio 13.6). La diagonal *d*=1 corresponde al caso básico, y después solo tienen sentido las diagonales impares, pues en cada etapa se consumen *2* cubos de comida. Así. cuando *i* varía de 1 a *n-d y j= i+d,* el número de cubos considerados, *j - i* + 1 siempre es par.

El algoritmo, que tiene coste $\Theta(n^2)$ tanto en espacio como en tiempo, es el siguiente:

```
fun vacas-pd(P[1..n] de real+) dev
máximo: real+
var vacas[1..n, 1..n] de real+

{inicialización}
para i = i hasta n - 1 hacer
  vacas[i, i + 1] := máx(P[i], P[i + 1]);
fpara

{rellenar matriz}
para d = 3 hasta n - 1 paso 2 hacer
  para i = 1 hasta n - d hacer
    j := i + d
    si P[j] > P[i+1] entonces
      como-i := vacas[i+1, j - 1];
    sino
      como-i := vacas[i+2, j];
    fsi

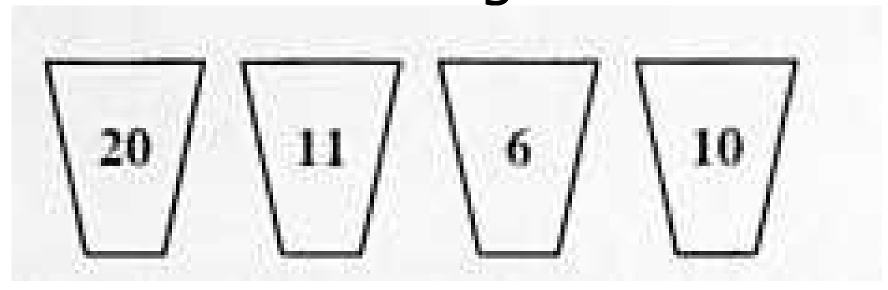
    si P[i] > P[j -1] entonces
      como-j := vacas[i+1, j-1];
    si no
      como-j := vacas[i, j-2];
    fsi
    vacas[i, j] := máx(P[i] + como-i, P[j] +
    como-j);
  fpara
fpara
```

```
máximo := vacas[1,n];
ffun
```

Apartado (c)----------------------------------

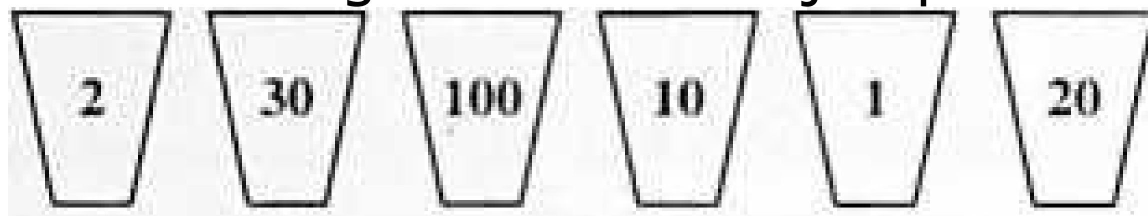
Si *Listilla* comienza comiendo, puede obligar a que *Decoradora,* siga la estrategia que siga, siempre tenga que comer en posiciones pares o impares. En consecuencia, si la suma de las cantidades de posiciones pares fuera mayor que la suma de las posiciones impares, *Listilla* podría empezar comiendo el cubo *n* (par). A continuación *Devoradora* tendría que comer el cubo 1 o el cubo *n*−1, pero en cualquier caso, dejaría un cubo par en uno de los extremos, que podría comer *Listilla.* Y así sucesivamente. Lo único que tiene que hacer *Listilla* es, por tanto, sumar la cantidad de comida en las posiciones pares e impares, y empezar por el extremo correspondiente, para en lo sucesivo seguir comiendo los cubos disponibles con dicha paridad. El algoritmo, por supuesto, es lineal.

Sin embargo, esta estrategia no es siempre óptima. Por ejemplo, si la situación inicial es la siguiente



Listilla decidiría comer los cubos en posiciones impares, aunque esa no sea la mejor opción. Podemos mejorar la estrategia haciendo que *Listilla* vuelva a

considerar comerse los cubos en posiciones pares o impares cada vez que le toque comer. Sin embargo, esta estrategia sigue sin ser óptima, como muestra el siguiente contraejemplo:



Con la estrategia *Listilla* comería 2+100+10 = 112. mientras que empezando por el último cubo podría comer 20+1 + 100 = 121.

El siguiente algoritmo, de complejidad lineal en tiempo y constante en espacio, implementa la estrategia mejorada.

```
fun vacas-par-impar(P[1..n] de real+) dev
cuánto : real +

{calcular sumas de posiciones pares e impares}
impares := 0 ;
pares := 0
para i = 1 hasta n hacer
si impar(i) entonces impares := impares + P[í]
sino pares := pares + PL]
fsi
fpara
izq := 1 : der := n ; cuánto := 0
mientras izq < der hacer
{come Listilla, hay un número par de cubos}
si impares > pares entonces {come impar}
si impar (Le/) entonces
cuánto := cuánto + P [/£<:/] ; impares :=
impares — P[izq]
izq := izq + 1
si no
cuánto := cuánto + P\der\ ; impares := impares
— P[der]
der := der — 1
fsi
si no { come par }
```

```
si impar(íz^) entonces
cuánto := cuánto + P[der] ; pares := pares —
P[der}
der := der — 1
si no
cuánto := cuánto +      : pares := pares — Ffízg]
iZq '■= izq + 1
fsi
fsi
{ come Devoradora}
si P|í;r/| > P[der] entonces { come por la
izquierda )
si imparO'.v/) entonces impares := impares —
P[izq\
si no pares := pares — /J[icr/|
fsi
izq \= izq + 1
si no ( come por la derecha )
si impar(íZer) entonces impares := impares —
P[der]
si no pares := pares — P[der]
fsi
der der — i
fsi
fmientras
ffun
```

13.18. Combinatoria

(a) Diseñar un algoritmo para calcular el número combinatorio $\binom{n}{r}$ basándose en la igualdad

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$$

que se tiene cuando $n > r > 0$. El coste en tiempo del algoritmo debe estar en $\Theta(nr)$ *y* el espacio adicional en $\Theta(r)$.

(b) Se dispone de *n* objetos que es necesario ordenar empleando las relaciones < y =. Por ejemplo, con 3 objetos se tienen 13 ordenaciones posibles:

$$a = b = c \quad a = b < c \quad a < b = c$$
$$a = c < b \quad b < a = c \quad b < a < c$$
$$c < a = b \quad c < a < b \quad c < b < a$$
$$a < b < c \quad a < c < b$$
$$b < c < a \quad b = c < a$$

Desarrollar un algoritmo para calcular en función de *a* el número de ordenaciones posibles, invirtiendo un tiempo que esté en $\Theta(n^2)$ y un coste en espacio auxiliar que esté en $\Theta(n)$.

-------------------------------Solución:-------------------------

Apartado (a)

El caso básico para calcular $\binom{n}{r}$ se obtiene para *r*=0 *y r*=*n*, *y* entonces $\binom{n}{r}$ = 1.

El caso recursivo se basa en la igualdad del enunciado entre números combinatorios

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$$

que se da directamente para números naturales verificando $n > r > 0$. pero que

es también cierta para $r \geq n$ si definimos $\binom{n}{r} = 0$ cuando $r > n$. Esto nos permite calcular iterativamente los números combinatorios, rellenando la Tabla 13.2 por filas.

	0	1	2	…	r
0	1	0	0	…	0
1	1	1	0	…	0
2	1	2	1	…	0
⋮	⋮	⋮	⋮	⋱	⋮
n	1	n	$\binom{n}{2}$	…	$\binom{n}{r}$

Tabla 13.2: Triángulo de Pascal.

Dejando aparte los casos básicos, para calcular cada entrada *(i,j)* en la tabla se necesitan las entradas (i−1, *j*-1) e *(i*-1, j) de la fila anterior, por lo que el espacio adicional se puede reducir a un vector que se rellena de derecha a izquierda (como, por ejemplo, en el Ejercicio 13.4). De esta forma se llega al siguiente algoritmo, cuyos costes son los requeridos.

{r <n}

```
fun pascal(n, r : nat) dev c: nat
var C[0..r] de nat

C[0]:= 1;
C[1..r]:= [0];
  para i = 1 hasta n hacer
    para k = r hasta 1 paso - 1 hacer
      C[k] := C[k]+ C[k-1];
    fpara
  fpara
c := C[r];
ffun
```

$\{c= \binom{n}{r}\}$

Apartado (b)----------------------------

Definimos la función

N(n) = número de ordenaciones posibles.

Obviamente, tenemos N(0) = 1, N(1) = 1, *N(2)* = 3. Veamos cómo definir inductivamente *N(n)* a partir de los valores $\{N(k) \mid 0 \leq k < n\}$.

Si tenemos una ordenación entre n objetos $a_1,\ldots,a_n$, distinguirnos la posición *k* (con $1 \leq k \leq n$)

tal que

$$a_1 = \cdots = a_k < a_{k+1} \leqq \cdots \leqq a_n.$$

En esta situación, el orden entre los elementos $a_1,\ldots,a_k$ no importa porque se relacionan mediante la igualdad; pero como los elementos que son iguales entre sí y menores que los demás sí importan (es decir, pueden ser cualesquiera de los dados), tenemos $\binom{n}{k}$ posibilidades para esta parte de la ordenación. Los restantes *n - k* elementos vienen determinados por la elección anterior, pero entre ellos hay *N*(n - *k)* formas diferentes de ordenarlos, por lo que el total de posibilidades es $\binom{n}{k}N(n-k)$, *suponiendo k fijo.*

Naturalmente, haciendo variar *k* entre 1 y *n* se obtienen todas las posibles ordenaciones diferentes, por lo que se tiene

$$N(n) = \sum_{k=1}^{n} \binom{n}{k} N(n-k).$$

Se puede comprobar inmediatamente

que, en particular, con esta fórmula se obtienen, en efecto, N(1) = 1, N(2) = 3 y N(3) = 13.

Para la implementación, observarnos que para calcular *N(i)* necesitamos conocer $\binom{i}{k}$ para 1 < *k* < *i* con lo que vamos a adaptar el algoritmo pascal anterior, con *r* = *n*. calculando *N(i)* justo después de haber calculado todos los números combinatorios $\binom{i}{k}$ de la fila *i*.

```
fun ordenaciones1 (n: nat) dev número : nat
var C[0..n], N[0..n] de nat

C[0] := 1;
N[0] := 1;
C[1..n] := [0];
para i = 1 hasta n hacer
  para k = n hasta 1 paso - 1 hacer
    C[k] := C[k] + C[k-1];
  fpara

  N[i] := 0;
  para k = 1 hasta i hacer
    N[i] := N[i] + C[k] * N[i-k];
  fpara
fpara
número := N[n];
ffun
```

Teniendo en cuenta que hemos considerado el caso particular *r* = *n* del algoritmo pascal anterior, el siguiente algoritmo es una posible optimización en la que se juntan los dos bucles para *k*. de

modo que se van calculando incrementalmente los valores *N(i)* al mismo tiempo que se van calculando los números combinatorios (¿) de la fila *i.*

```
fun ordenaciones2(n : nat) dev número: nat
var C[0..n], N[0..n] de nat

C[0] := 1;
N|0] := 1;
para i = 1 hasta n hacer
    C[i] := 1;
    N[i] := 1;
    para k = i - 1 hasta 1 paso - 1 hacer
        C[k] := C[k] + C[k-1];
        N[i] := N[i] + C[k] *N[i-k];
    fpara
fpara
número := N[n];
ffun
```

En ambos casos, los costes en tiempo y espacio adicional son los deseados: *&(n~)* y 9(n), respectivamente

Capítulo 14

14. VUELTA ATRÁS

ARBOLES DE EXPLORACION

En algunas ocasiones, las técnicas vistas en los tres capítulos anteriores no son aplicables, y el último recurso que nos queda para resolver nuestro problema es aplicar la "fuerza bruta", es decir, realizar una búsqueda exhaustiva por el espacio de posibles soluciones hasta encontrar una que satisfaga los criterios exigidos, o constatar que tal solución satisfactoria no existe. Sin embargo, esta búsqueda exhaustiva resultara impracticable si el cardinal del conjunto de soluciones posibles es muy grande, lo cual ocurre muy a menudo. Es. por tanto, imprescindible intentar estructurar el espacio a explorar, tratando de descartar en bloque posibles soluciones no satisfactorias.

Consideraremos problemas cuyas soluciones sean construibles por etapas. De esta forma, una solución será expresable en forma de n-tupla $(x_1,\ldots,x_n)$ donde cada xi ∊ Si representa la decisión tomada en la etapa i-esima. de entre un conjunto finito de alternativas. Además, una solución habrá de minimizar, maximizar o simplemente satisfacer una cierta *función criterio.* Se establecen entonces dos categorías de restricciones:

- las **restricciones explícitas,** que indican los conjuntos 5, a los que pertenecen cada una **de las** componentes de una tupla solución;
- las **restricciones implícitas,** que son las relaciones que se han de establecer entre las componentes de la tupla solución para

satisfacer la función criterio.

El espacio de soluciones estará formado por el conjunto de tuplas que satisfacen las restricciones explícitas. y se puede estructurar como un *árbol de exploración,* donde en cada nivel se toma la decisión de la etapa correspondiente. Así, se denomina *nodo estado* a cualquier nodo del árbol de exploración, correspondiente a una tupla parcial o a una tupla completa, y satisfaciendo las restricciones explícitas; mientras que *nodos solución* serán los correspondientes a las tuplas completas que satisfacen las restricciones implícitas.

Un elemento adicional imprescindible para la aplicabilidad de esta técnica lo constituyen las denominadas *funciones de poda* o *tests de factibilidad* (obtenidas a partir de la función criterio) que permiten determinar cuándo una tupla parcial nunca va a conducir a una solución satisfactoria, por lo que es inútil seguir buscando a partir de ella.

Una vez definido el árbol de exploración, el algoritmo para resolver el problema consistirá en realizar un recorrido del árbol en cierto orden, hasta encontrar la primera solución, o bien recorrer el árbol completo (salvo las zonas podadas) para obtener todas las soluciones o la solución óptima deseada. Durante el proceso, para cada nodo se irán generando sus sucesores; así denominaremos *nodos vivos* a aquellos para los cuales todavía no se han generado todos sus hijos; *nodos en expansión* a aquellos cuyos hijos están siendo generados; y *nodos muertos* a aquellos que no pueden ser expandidos, bien porque no hayan superado el test de factibilidad, o bien porque todos sus hijos ya han sido generados.

De entre las posibles formas de recorrer el árbol de exploración se destacan dos, que dan lugar a las técnicas aplicadas en este capítulo y en el siguiente:

- **Vuelta atrás:** el recorrido se realiza en profundidad, de forma que los nodos vivos se gestionan mediante una ***pila***. El método resulta sencillo y eficiente en espacio.
- **Ramificación y poda:** corresponde a una búsqueda más "inteligente", donde en cada momento se expande el nodo vivo más "prometedor", de forma que los nodos vivos se gestionan mediante una ***cola con prioridad***.

El coste en el caso peor de todos estos algoritmos de búsqueda exhaustiva está en el orden del tamaño del espacio de soluciones, que suele ser cuando menos exponencial. Su utilidad práctica reside en la efectividad de las funciones de poda que se apliquen: cuanto más elaboradas sean, más nodos no factibles se detectarán y mayor proporción de árbol se podará. Sin embargo, si se vuelven demasiado sofisticadas, su coste de aplicación (a cada nodo del árbol) puede no llegar a compensar la poda obtenida. Este tipo de cuestiones difícilmente son analizables de forma teórica, y solo se puede llegar a un criterio adecuado en base a medidas empíricas sobre cada aplicación concreta.

En las soluciones expuestas en este capítulo y el siguiente, se proponen diversas funciones de poda que, intuitivamente, parecen ser beneficiosas, pero no se realiza ningún análisis sobre su efectividad práctica.

ESQUEMAS PARA LA VUELTA ATRÁS

Cuando se realiza una búsqueda en profundidad y se llega a un nodo muerto, hay

que deshacer la última decisión tomada, para optar por la siguiente alternativa (como cuando en un laberinto se llega a un callejón sin salida). Por eso a esta técnica se la denomina ***vuelta atrás*** (*backtracking* en inglés) y la forma más sencilla de expresarla es mediante un algoritmo recursivo, donde la vuelta atrás es automáticamente controlada por el mecanismo que implementa la recursión.

14.2.1. Esquema general

El esquema general recursivo de vuelta atrás que vamos a seguir es el siguiente:

```
proc vuelta-atrás(sol: tupla, e k : nat)
 preparar-recorrido-nivel (k)
  mientras ¬ último-hijo-nivel (k) hacer
   sol[k] := siguiente-hijo-nivel(k)
    si es-solución?(sol, k) entonces
      tratar-solución (sol)
    si no
      si es-completable?(sol, k) entonces
        vuelta-atrás(sol, k+1)
      fsi
    fsi
  fmientras
fproc
```

Suele suceder que los árboles de exploración se definen de tal forma que <u>los nodos solución solo se encuentran en las hojas</u>. El esquema anterior expresa esta condición. Si no fuera ese el caso, simplemente habría que considerar la posibilidad de expandir el nodo tras tratar la solución.

El esquema dado va a <u>obtener *todas* las soluciones</u>. Si solo se desea encontrar *una* solución. basta añadir un parámetro booleano *éxito* (inicializado a falso), que tome el valor cierto al encontrar la primera solución. La

variable *éxito* controlará el bucle de generación de los hijos del nodo en expansión, y será un parámetro de entrada/salida que haga finalizar los bucles en llamadas recursivas pendientes (véase, por ejemplo, el Ejercicio 14.3).

14.2.2. Marcaje

La aplicación del test de factibilidad conlleva en ocasiones un cálculo elaborado que, en la mayoría de los casos, se puede optimizar por el método de asociar a cada nodo cierta cantidad de información que corresponde a "cálculos parciales" de dichos tests. De esta forma se incrementa el coste en memoria a cambio de reducir el tiempo de ejecución. A esta técnica la denominamos ***marcaje*** y la forma más habitual de realizarla es mediante parámetros adicionales *(marcadores)* de entrada/salida que equivalen a variables globales y por tanto suponen un pequeño incremento del coste en espacio.

```
proc vuelta-atrás-marcaje(sol: tupla, e k:
nat, m: marcador)
 preparar-recorrido-nivel (k)
  mientras ¬ último-hijo-nivel(k) hacer
    sol[k]:= siguiente-hijo-nivel(k)
    m:= marcar(m, sol[k])
     si es-solución?(sol, k) entonces
       tratar-solución (sol)
     si no
       si es-completable?(sol, k, m)
       entonces
         vuelta-atrás-marcaje(sol, k+1, m)
       fsi
     fsi
    m:= desmarcar(m, sol[k])
  fmientras
fproc
```

Obsérvese que. en principio, hay que desmarcar antes de pasar al hijo siguiente, pues hay que restaurar la situación inicial correspondiente al nodo padre. Una alternativa sería utilizar variables locales, de forma que se reserva nuevo espacio con cada llamada recursiva (para cada nodo recorrido). Si bien ello incrementa considerablemente el coste en espacio, puede ser necesario cuando desmarcar es inviable (véase, por ejemplo, el Ejercicio 14.21).

14.2.3. Optimización

Para tratar problemas de optimización tenemos que modificar los esquemas anteriores, de forma que se almacene la mejor solución encontrada hasta el momento. Así. a la hora de tratar una nueva solución, se comparará con la que se tiene almacenada. Para realizar esta comparación de la forma más eficiente posible, es preciso almacenar junto con la mejor solución su valor asociado. Además, para facilitar el cálculo del valor de cada solución alcanzada se incorpora como marcador el valor (parcial) de la tupla parcial. Por último, los problemas de optimización admiten un mecanismo adicional de poda, que se realiza cuando se puede asegurar que ninguno de los descendientes del nodo a expandir puede llegar a alcanzar una solución mejor que la mejor encontrada hasta ese momento. Así. en un problema de minimización. una cota inferior o estimación de la mejor solución alcanzable desde un nodo sirve para podarlo si la estimación es ya mayor que el valor asociado a la mejor solución encontrada hasta el momento.

El esquema correspondiente es el siguiente:

```
proc vuelta-atrás-optimización (sol:
```

```
tupla, e k: nat, valor: valor, sol-mejor:
tupla, valor-mejor: valor)
preparar-recorrido-nivel(k)
mientras ¬ último-hijo-nivel(k) hacer
    sol[k] := siguiente-hijo-nivel(k)
    valor := actualizar(valor, sol, k)
    si es-solución?(sol, k) entonces
        si mejor(valor, valor-mejor)
        entonces
            sol-mejor:= sol ;
            valor-mejor:= valor
        fsi
    si no
        si es-completable?(sol, k) ∧ es-
        prometedor?(sol, k, valor, valor-
        mejor) entonces
            vuelta-atrás-optimización(sol, k+1,
            valor, sol-mejor, valor-mejor)
        fsi
    fsi
    valor:= desactualizar(valor, sol, k)
fmientras
fproc
```

EJERCICOS RESUELTOS V.A.

14.1. Calcular palabras

Dadas *n* letras, todas diferentes, y *m<n*, diseñar un algoritmo que calcule todas las palabras con *m* letras diferentes escogidas entre las n dadas.

--------------------------Solución------------------------

Lo que pide el ejercicio es encontrar las variaciones de *n* elementos, tomados de *m* en *m*.

Supongamos que las letras son {1,…,n}. Las soluciones del algoritmo serán palabras de *m* letras, que representamos con tuplas $(x_1, ..., x_m)$, donde *x, es* la i-ésima letra de la palabra. Las soluciones tienen que cumplir como restricciones explícitas que se utilicen letras válidas, y como restricciones implícitas que no haya letras repetidas.

El árbol de exploración, con *m* niveles, es de la forma mostrada en la Figura 14.1

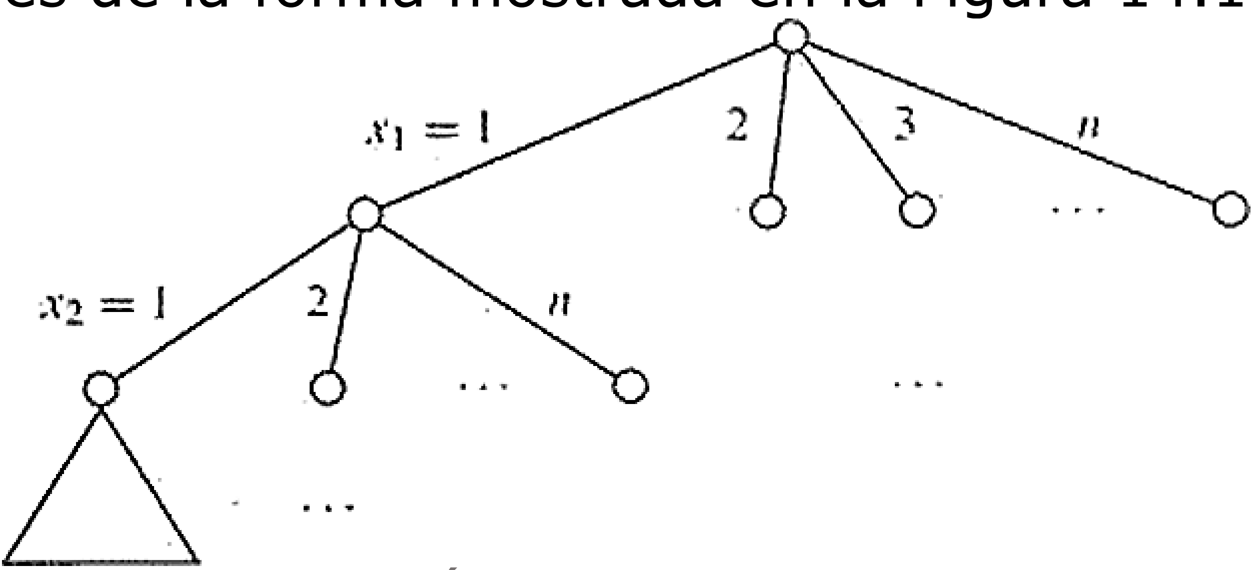


Figura 14.1: Árbol de exploración para el problema de las variaciones.

No necesitamos explorar todo el árbol, pues podemos podar nodos en el árbol cuando ya hayamos repetido letras en la solución parcial que vamos formando. En

cada paso, al extender la solución asignando a x_k la letra *j*, por ejemplo, comprobaremos que *j* no aparece ya en $(x_1,\ldots,x_{k-1})$, suponiendo que no hay repeticiones en $(x_1,\ldots,x_{k-1})$.

El algoritmo es el siguiente:

```
proc variacionesVa1 (e n: nat+,
sol[1..m] de 1..n,e k: 1..m)

  para j=1 hasta n hacer
    sol[k]:= j;
    si noRepetida?(sol, k) entonces
      si k = m entonces
        imprimir(sol); {es una solución}
      si no
        variacionesVa1 (n, sol, k+1);
      fsi
    fsi
  fpara
fproc
```

La función noRepetida?(*sol*, *k*) comprueba si la letra *sol*[*k*] aparece o no en *sol*[1..*k*-1]:

```
fun noRepetida?(sol[1..m] de nat,
k:1..n) dev respuesta: bool

  i:= 1;
  mientras sol[i] ≠ sol[k] hacer
    i := i+1;
  fmientras
  respuesta:= (i = k);
ffun
```

Podemos evitarnos el bucle de la función noRepetida? (de coste lineal) mediante la *técnica de* ***marcaje*** (Sección 14.2.2). Tendremos como *marcador* un vector

usada[1..n] de booleanos. que indicará si la correspondiente letra ha sido incluida en la solución parcial o no:

$$\forall i: 1 \leq i \leq n: usada[i]$$
$$\Leftrightarrow i \text{ aparece en } sol\ [1..k].$$

Así. para añadir una nueva letra, esta no debe estar *usada;* cuando añadamos una letra a la solución parcial se marcará como usada, y cuando volvamos atrás y se deshaga la solución la desmarcaremos.

El algoritmo con marcadores el siguiente:

```
proc variaciones-va2(e n:nat⁺, sol[1..m] de 1..n, e k:1..m,
usada[1..n] de bool)

  para j=1 hasta n hacer
    si ¬usada[j] entonces
      sol[k] := j;
      usada[j] := cierto;      {marcar}
      si k=m entonces
        imprimir(sol);
      si no
        variaciones-va2(n, sol, k+1, usada);
      fsi
      usada[j] := falso;      {desmarcar}
    fsi
  fpara
fproc
```

El procedimiento principal que inicializa la variable *usada* y que hace que se muestren todas las soluciones, llamando al algoritmo recursivo de vuelta atrás, es el siguiente:

```
proc variaciones(e n : nat⁺)
var sol[1..m] de 1..n, usada[1..n] de
bool

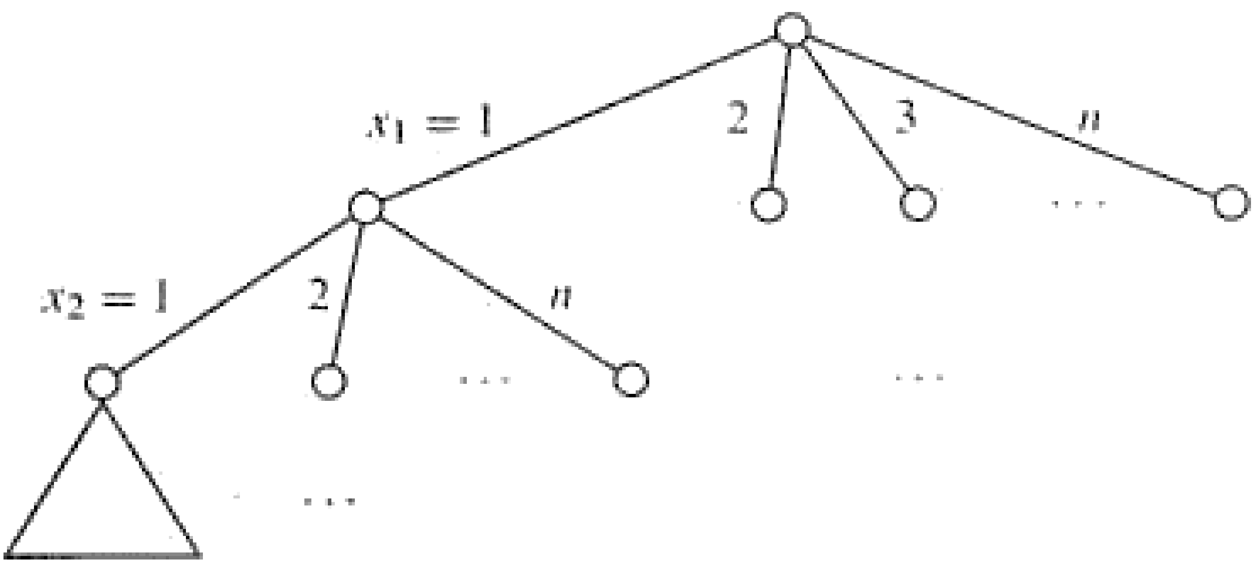
    usada[1..n]:= [falso];
    variacionesVa2(n, sol,1, usada);
fproc
```

14.2. Parejas "estables"

La agencia matrimonial "Su Media Naranja" asegura *matrimonios estables* a sus clientes. Para ello dispone de información almacenada en 2 matrices de números naturales *H[1..n,* 1..*n*] y *M[1..n,* 1..*n*] tales que *H[i, j]* indica la preferencia de un hombre *i* por una mujer *j* y *M[i, j]* la preferencia de una mujer *i* por un hombre *j,* para *i* y *j* entre 1 y *n.* Establecida una asignación de *n* parejas, si existe algún hombre y alguna mujer que, sin estar emparejados entre sí, se prefieren mutuamente a sus respectivas parejas, entonces la asignación es *inestable;* si no se da tal caso, la asignación es *estable.* Desarrollar un algoritmo que encuentre todos los emparejamientos *estables*.

--------------------------Solución------------------------

Si numeramos los hombres y las mujeres de 1 a *n* podemos representar cada emparejamiento por medio de una tupla $(x_1,\ldots,x_n)$ donde xi es el hombre asignado a la mujer *i.* Se tiene que cumplir que a cada mujer le ha sido asignado un hombre de los existentes y que cada hombre ha sido asignado a una sola mujer. Es decir, una posible solución es una *permutación* de $\{1,\ldots,n\}$. En consecuencia, el árbol de exploración es similar al de la Figura 14.1. con *n* niveles y *n* hijos por nodo.



Para comprobar que un hombre no es asignado a más de una mujer utilizaremos como marcador un vector *asignado*[1 ..*n]* de booleanos que indica si un hombre ya ha sido asignado a una mujer en la solución parcial calculada hasta el momento.

Cada vez que se asigna un hombre a una mujer podemos comprobar si los emparejamientos realizados hasta el momento son estables, lo que permite podar el árbol en el caso en que se encuentre que los emparejamientos realizados son ya inestables.

El procedimiento recursivo que imprime todos los emparejamientos estables es el siguiente:

```
proc parejasVa(e H[1..n, 1..n], M[1..n, 1..n] de
nat, sol[1..n] de 1..n, e k:1..n, asignado[1..n] de
bool)

  para hombre = 1 hasta n hacer
    si ¬asignado[hombre] entonces
      sol[k]:= hombre;
      asignado[hombre]:= cierto;  {marcar}
      si estable?[H, M, sol, k) entonces
        si k=n entonces
          imprimir(sol);
        si no
          parejasVa(H, M, sol, k+1, asignado);
        fsi
      fsi
      asignado[hombre]:= falso;   {desmarcar}
```

```
    fsi
  fpara
fproc
```

Como se llega al nivel *k* después de comprobar la factibilidad en el nivel *k*-1, sabemos que la tupla (x_1,…,x_{k-1}) es estable, por lo que al añadir la decisión sobre x_k basta comprobar que se mantiene la propiedad con respecto a las anteriores parejas.

```
fun estable? (H[1..n, 1..n], M[1..n, 1..n] de nat,
sol[1..n] de 1..n, k: 1..n) dev respuesta: bool

respuesta:= cierto;
i := 1;
mientras i<k ∧ respuesta hacer
   respuesta:= (M[k,sol[i]] ≤ M[k, sol[k]] v
   H[sol[i], k] ≤ H[sol[i], i]) ∧ (M[i,sol[k]] ≤
   M[i,sol[i]] v H[sol[k],i] ≤ H[sol[k],k]);
   i:= i+1;
fmientras
ffun
```

El procedimiento principal es:

```
proc parejas(e H[1 ..n, 1..n], M[1 ..n, 1..n]
de nat)
var sol[1..n] de 1..n, asignado[1 ..n] de bool

  asignado[1..n]:= [falso];
  parejasVa(H, M, sol, 1, asignado);
fproc
```

ANÁLISIS DE LA EQUIVALENCIA DE LOS PROBLEMAS DE PAREJAS ESTABLES/AFINES:

PROBLEMA 1 (Emparejamiento estable):
- El objetivo de este problema es encontrar **emparejamientos estables** entre hombres y mujeres, donde no exista una pareja (no asignada entre sí) que prefiera mutuamente estar juntos más que con sus respectivas parejas actuales. Este es un problema típico de **estabilidad de emparejamientos**, y el algoritmo clásico para resolverlo es el **algoritmo de Gale-Shapley** (también conocido como el algoritmo de emparejamiento estable).

PROBLEMA 2 (Mesa redonda con afinidades):
- El **objetivo** de este problema es sentar a hombres y mujeres en una mesa redonda, alternando géneros, con la restricción de que ninguna persona puede sentarse junto a su pareja habitual, y maximizando el bienestar total en función de las afinidades mutuas entre las personas adyacentes. Este es un problema de **optimización combinatoria**, donde la búsqueda de una solución óptima (maximización del bienestar) implica probar distintas combinaciones de personas sentadas, respetando las restricciones.

¿Son equivalentes?
Los problemas NO son equivalentes. Aquí están las razones:

1. **Objetivo**:
- En el **problema 1**, el objetivo es encontrar una asignación de parejas **estables**, es decir, una solución donde no existan desviaciones preferenciales.
- En el **problema 2**, el objetivo es maximizar una **función de afinidad** mientras se respetan las restricciones de alternancia y de no sentarse junto a la pareja habitual.

2. **Naturaleza del problema**:
- El **problema 1** es más un problema de estabilidad en los emparejamientos, que se resuelve con una búsqueda secuencial y condiciones de estabilidad.
- El **problema 2** es un problema de optimización que implica maximización de afinidades en una disposición circular, lo que lo convierte en un problema más complejo en términos de combinaciones y valores a

maximizar.

¿CÓMO RESOLVER CADA PROBLEMA?

PROBLEMA 1 (Emparejamiento estable):
- El algoritmo clásico para resolver este tipo de problemas es el algoritmo de Gale-Shapley, que no necesita ni vuelta atrás ni ramificación y poda, ya que se trata de un algoritmo voraz que garantiza una solución estable en tiempo polinómico. No explora todas las combinaciones, sino que va asignando parejas secuencialmente según preferencias.

Veredicto: Este problema no se resuelve mediante **vuelta atrás** ni **ramificación y poda**, ya que el algoritmo de emparejamiento estable tiene un enfoque directo y eficiente.

PROBLEMA 2 (Mesa redonda con afinidades):
- Este problema involucra la búsqueda de la combinación óptima que maximice el bienestar total, cumpliendo con las restricciones impuestas. Aquí tienes varias combinaciones posibles para asignar los asientos, y el problema es encontrar aquella que maximice la suma de afinidades.

Para resolverlo, puedes optar por ramificación y poda. Este enfoque es adecuado porque permite explorar las combinaciones de asientos y descartar de manera temprana aquellas configuraciones que no pueden llevar a una solución óptima (por ejemplo, si la afinidad acumulada en una rama es ya menor que la mejor encontrada hasta el momento). Este método es más eficiente que la **vuelta atrás**, ya que puedes descartar muchas combinaciones que no cumplirán con las restricciones o no serán óptimas.

Veredicto: El **problema 2** debería resolverse con **ramificación y poda**, ya que permite maximizar el bienestar mientras se descartan ramas no prometedoras. La **vuelta atrás** también sería una opción válida, pero menos eficiente.

Conclusión:
- Los problemas no son equivalentes, ya que tienen diferentes objetivos y estructuras.
- El **problema 1** se resuelve mejor con el algoritmo de Gale-Shapley (ni vuelta atrás ni ramificación y poda).
- El problema 2 es un problema de optimización que

puede abordarse mediante ramificación y poda** para mejorar la eficiencia en la búsqueda de la solución óptima.

14.3. Isomorfos

Diseñar un algoritmo que verifique si 2 grafos dados son **ISOMORFOS**.

Dos grafos dirigidos *G y G'* son *isomorfos* si existe una función biyectiva *f* entre los vértices de ambos grafos de forma que ⟨x, y⟩ es una arista de G si y solo si ⟨*f(x), f* (y)⟩ es una arista de G'.

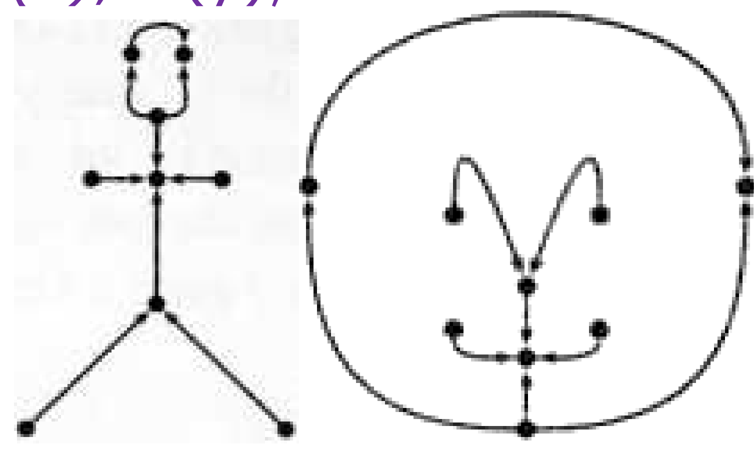


Figura 14.2: Dos grafos isomorfos.

--------------------------Solución-----------------------

Vamos a suponer para empezar que el número total de vértices y de aristas en cada grafo coincide, ya que en otro caso los grafos no son isomorfos. En estas condiciones, suponemos que el conjunto de vértices de cada grafo es {1,…,n}.

Como una solución es una función biyectiva de {1,…,n} a {1,…,n}. podemos representarla como una tupla $(x_1,\ldots,x_n)$, donde xi ∈ {1,…,n} es el vértice de G' que la función asocia al vértice *i* de G, que es una permutación del conjunto {1,…,*n*}. Además de ser biyectiva, la función solución tiene que respetar las aristas, es decir.

está-arista?(i, *j, G)* ⇔ está-arista? (xi,*xj,G').*

En particular, los grados de entrada y de salida de un vértice *i* y de su imagen xi tienen que coincidir.

Suponemos que los grafos vienen

representados mediante sus matrices de adyacencia, a partir de las cuales calcularemos el grado de entrada y de salida de cada vértice mediante el siguiente algoritmo:

```
fun obtenerGrados(G : grafo[n]) dev
⟨E[1..n], S[1..n] de 0..n⟩

E[1..n]:= [0];
S[1..n] := [0];
para i=1 hasta n hacer
  para j=1 hasta n hacer
    si G[i, j] entonces
      E[j] := E[j] + 1;
      S[i] := S[i] + 1;
    fsi
  fpara
fpara
ffun
```

El coste de obtener los grados está en $\Theta(n^2)$.

El árbol de exploración es similar al de la Figura 14.1. con *n* niveles. Podemos podar un nodo del árbol cuando o bien se intenta utilizar un vértice de *G′* que ya está en la imagen de la función parcialmente construida (para ello utilizaremos como marcador un vector *usado*[1..n] de booleanos), o bien no **coinciden los** grados de entrada o salida, o bien existe una arista que no se respeta, es decir, ya hemos definido x_i *y* x_j y tenemos una arista en G entre *i y j* pero no existe arista en *G′* entre x_i y x_j, o viceversa.

El algoritmo tiene que terminar cuando

se sabe que los grafos son isomorfos, es decir, al encontrar una función biyectiva válida. Utilizamos una variable booleana *éxito* cuyo valor inicial es falso y se convierte en cierto al encontrar la primera solución.

```
proc isomorfosVa(e G, G': grafo[n], e
E[1..n], S[1..n], E'[1..n], S'[1..n] de 0..n,
sol[1..n] de 1..n, e k: 1..n, usado[1..n] de
bool, éxito: bool)

vértice := 1;
  mientras vértice≤n ∧ ⌐éxito hacer
    si ⌐usado[vértice] entonces
      sol[k]:= vértice;
      usado[vértice]:= cierto; {marcar}
      si (E[k] = E'[sol[k]] ∧ S[k] = S'[sol[k]])
      ∧ hayAristas?(G, G', sol, k) entonces
        si k=n entonces
          éxito:= cierto;
        si no
          isomorfosVa(G, G', E, S, E', S', sol, k+1, usado,
          éxito);
        fsi
      fsi
      usado[vértice]:= falso; {desmarcar}
    fsi
    vértice:= vértice + 1;
  fmientras
fproc
```

La función hayAristas? comprueba si se respetan las aristas, suponiendo que desde 1 hasta *k*-1 ya se han comprobado:

```
fun hayAristas?(G, G': grafo[n], sol[1..n] de
nat, k: 1..n) dev respuesta: bool

i:= 1;
respuesta:= cierto;
```

```
mientras respuesta ∧ i<k hacer
    respuesta:= (G[i, k] = G'[sol[i], sol[k]]) ∧
    (G[k, i] = G'[sol[k], sol[i]]);
    i:= i+1;
fmientras
ffun
```

La función principal que realiza la inicialización y llama al procedimiento recursivo es la siguiente:

```
fun isomorfos?(G, G': grafo[n]) dev ⟨éxito:
bool, sol[1..n] de 1..n⟩
var usado[1..n] de bool, E[1..n], S[1..n],
E'[1..n], S'[1..n] de 0..n

usado[1..n]:= [falso];
⟨E, S⟩:= obtenerGrados(G);
⟨E',S'⟩:= obtenerGrados(G');
éxito := falso;
isomorfosVa(G, G', E, S, E', S', sol, 1, usado,
éxito);
ffun
```

14.4. Cuadrado latino

Un ***CUADRADO LATINO*** de tamaño *n* es un tablero de *n* x *n* posiciones, cada una de ellas está pintada de un color escogido entre *n* diferentes. No puede haber colores repetidos en ninguna de sus filas ni en ninguna de sus columnas.

Dados *n* colores diferentes:

(a) Escribir un algoritmo que imprima todos los cuadrados latinos posibles de tamaño *n.*

(b) Considerando que una solución es equivalente a todas las que se generen a partir de ella simplemente mediante la permutación de los colores, modificar el algoritmo del apartado anterior para que solo se obtenga un representante de cada clase de equivalencia.

-------------------------Solución:--------------------------

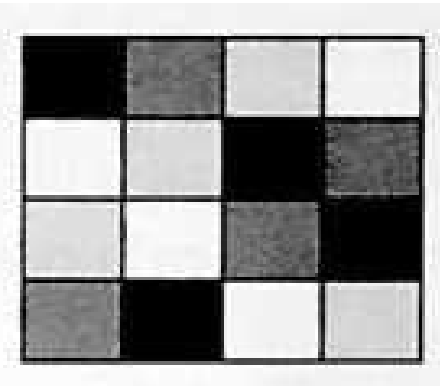


Figura 14.3: Cuadrado latino de tamaño 4.

	1	2		n
1	1	2	3 ···	n
2	$n+1$	$n+2$		$2n$
			⋱	
n				n^2

Figura 14,4: Posible numeración de las casillas de un cuadrado.

Apartado (a)------------------

Una posibilidad es dar las soluciones en forma de tupla $(x_1,\ldots,x_{n^2})$ donde x_i será el color con el que se ha pintado la casilla *i. y* el conjunto de colores es {1,…,n}. Para ello necesitamos numerar las casillas del tablero como muestra la Figura 14.4. y también funciones para pasar de un número de casilla *nc* a su correspondiente fila y columna:

fila(nc) = *((nc*-1) div *n)* + 1.
columna(nc) = *((nc*-1) mod n) + 1.

Las soluciones tienen que cumplir que utilizan colores válidos y que no hay dos casillas en la misma fila o columna con el mismo color.

El árbol de exploración es similar al del Ejercicio 14.1, con n^2 niveles y *n* hijos por nodo.

Ya que no podemos usar el mismo color dos veces en la misma fila o columna, utilizaremos marcadores para conocer en cada momento (en tiempo constante) si un color aparece ya en una fila o columna.

En concreto, tendremos matrices F[1..n, 1..n] y C[1..n, 1..n] de booleanos tales que

F[*i,color]* ⇔ en la fila *i* hemos usado el color *color.*

C[j,color] ⇔ en la columna *j* hemos usado el color *color.*

El algoritmo recursivo que imprime

todas las soluciones es el siguiente:

```
proc latinoVa1 (sol[1..n²] de 1..n, e k:
1..n, F[1..n, 1..n], C[1..n, 1..n] de bool)

fila:= fila(k) ;
columna:= columna(k);
para color=1 hasta n hacer
  si ⌐F[fila, color] ∧ ⌐C[columna, color]
  entonces
    sol[k]:= color;
    F[fila, color]:= cierto;
    C[columna, color]:= cierto    {marcar}
    si k=n² entonces
    imprimir(sol);
    si no
    latinoVa1 (sol, k+1, F, C);
    fsi
    F[fila, color]:= falso;
    C[columna, color] := falso   {desmarcar}
  fsi
fpara
fproc
```

El algoritmo principal con la inicialización correspondiente es:

```
proc latino1(e n: nat⁺)
var sol[1..n²] de 1..n, F[1..n, 1..n], C[1..n,
1..n] de bool

F[1..n, 1..n]:= [falso];
C[1..n, 1..n]:= [falso];
latinoVa1 (sol, 1, F, C);
fproc
```

Otra posibilidad es representar la solución como una matriz *sol*[1..n, 1..n] de forma que *sol*[*i, j*] indica el color asignado a la casilla situada en la fila *i* y la columna *j*. En este caso no necesitamos las funciones auxiliares fila ni columna, pero tenemos que decidir la casilla inicial, la final y, sobre todo, la forma de pasar de una casilla a la siguiente para pasar de nivel. Si recorremos las casillas del cuadrado latino de la misma manera que en la versión anterior, la casilla inicial es la $\langle 1,1\rangle$, la casilla final es la $\langle n,n\rangle$, y la casilla siguiente a la $\langle i, j\rangle$ es la $\langle i, j+1\rangle$ cuando $i \leq n$ y $j < n$ y la $\langle i+1, 1\rangle$ cuando $i < n$ y $j = n$.

El algoritmo queda como sigue:

```
proc latinoVa2(sol[1..n, 1..n] de 1..n, e i, j: 1..n, F[1..n, 1...n], C[1 ..n, 1..n] de bool)

  para color = 1 hasta n hacer
    si ¬F[i, color] ∧ ¬C[j, color] entonces
      sol[i, j] := color;
      F[i, color] := cierto;
      C[j, color]:= cierto;    {marcar}
      casos
           i=n ∧ j=n → imprimir(sol)
         □ i<n ∧ j=n → latinoVa2(sol, i+1, 1, F, C)
         □ i≤n ∧ j<n → latinoVa2(sol, i, j+1, F, C)
      fcasos
      F[i, color]:= falso;
      C[j, color]:= falso;     {desmarcar}
    fsi
  fpara
fproc
```

El correspondiente algoritmo principal en este caso es:

```
proc latino2(e n: nat+)
var sol[1..n, 1..n] de 1..n, F[1..n, 1..n], C[
1..n, 1..n] de bool

F[1..n, 1..n] := [falso];
C[1..n, 1..n] := [falso];
latinoVa2(sol, 1, 1, F, C)
fproc
```

Apartado (b)-------------------

Para evitar encontrar soluciones equivalentes es suficiente fijar de manera arbitraria pero única los colores de las casillas de la primera fila. Utilizando la segunda representación del apartado anterior la forma más sencilla de fijar el coloreado de la primera fila es utilizar al efecto la permutación identidad, *sol*[1,j]=j, lo que nos conduce al siguiente algoritmo principal:

```
proc latinoSinEquivalencias(e n: nat+)
var sol[1..n, 1..n] de 1..n, F[1 ..n. 1..n], C[
1..n, 1..n] de bool

  F[1..n, 1..n]:= [falso];
  C[1..n, 1..n]:= [falso];
  para j=1 hasta n hacer
    sol[1, j]:= j;
    F[1,j]:= cierto;
    C[j,j]:= cierto;
  fpara
  latinoVa2 (sol, 2, 1, F, C);
fproc
```

Nótese cómo la llamada inicial a la función recursiva comienza a pintar desde la casilla ⟨2, 1⟩.

14.5. Asignar maquinas

La República de Fanfanisflán acaba de abrir *n* factorías de tecnología punta, que requieren una instalación informática y personal que la dirija. El Gobierno Supremo les ha asignado n máquinas y otros tantos técnicos recién licenciados, que deberán ser distribuidos entre las fábricas.

Como no todas las máquinas son apropiadas para las necesidades de todas las fábricas, se dispone de una función booleana **apropiada?**(m, *f)* que indica si la máquina *m* es apropiada para la fábrica *f.*

Asimismo, no todos los técnicos conocen todas las máquinas, por lo que se dispone de la correspondiente función conoce?(t, *m)* que indica si el técnico *t* conoce la máquina *m.*

Finalmente, no todos los técnicos están dispuestos a trabajar en todas las fábricas (ya que algunas de ellas están relacionadas con la industria bélica), por lo que se dispone de la función aceptaría?(t, f) que indica si el técnico *t* estaría dispuesto a trabajar para la fábrica *f.*

Se pide encontrar, si la hubiera, alguna forma factible de asignar a cada fábrica una máquina apropiada a sus necesidades, y un técnico que conozca dicha máquina y acepte trabajar para la fábrica en cuestión.

--------------------------Solución:------------------------

Podemos expresar la solución como una tupla (x_1,…,x_n) *de pares* donde x_i= (m_i, t_i) indica que a la fábrica *i* se le han asignado la máquina m_i *y* el técnico t_i.

El árbol de exploración tiene la estructura de la Figura 14.5. donde cada nodo tiene n^2 hijos y hay *n* niveles.

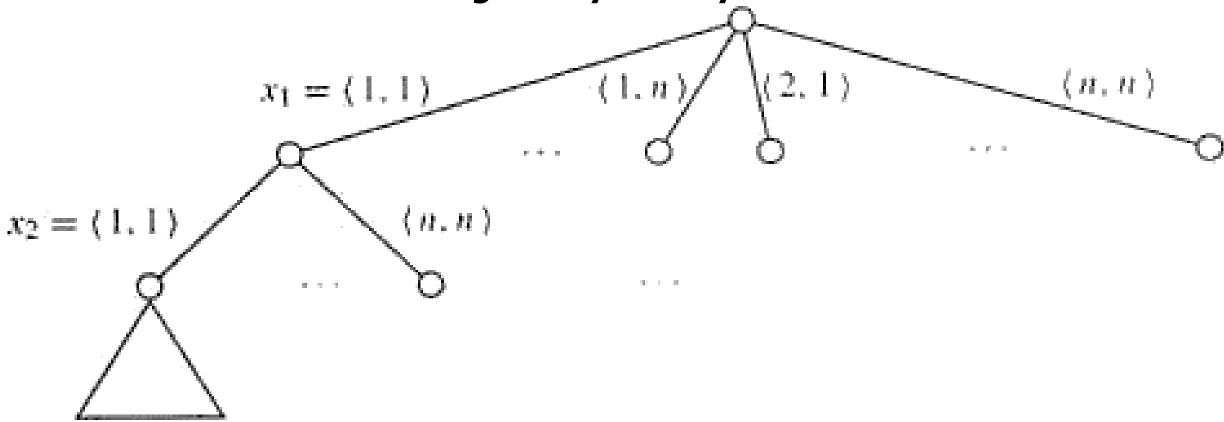


Figura 14.5: Árbol de exploración para el problema de las factorías.

Utilizamos marcadores para conocer qué máquinas y qué técnicos han sido ya asignados. De este modo un nodo no es factible (se puede podar) porque ya están previamente asignados la máquina o el técnico; o bien porque no es apropiada la máquina, o el técnico no conoce la máquina, o no acepta trabajar en la fábrica.

Un posible algoritmo es el siguiente, donde las diferentes partes del test de factibilidad se engloban todas juntas en la función aceptable?. Escribimos un versión que busca solo la primera solución (como en el Ejercicio 14.3):

tipos

pareja = **reg**

máquina: 1 ..*n*

técnico: 1..n

freg
ftipos

```
proc fábricasVa1(sol[1..n] de pareja, e k: 1..n, usadaMáq[1..n], usadoTéc[1..n] de bool, éxito: bool)

m:=1     {recorre las máquinas}
mientras m≤n ∧ ⌐éxito hacer
  t:=1;     {recorre los técnicos}
  mientras t≤n ∧ ⌐éxito hacer
    si aceptable?(k, m, t, usadaMáq, usadoTéc)
    entonces
      sol[k].máquina:= m;
      sol[k}.técnico:= t;
      usadaMáq[m]:= cierto;
      usadoTéc[t]:= cierto     {marcar}
      si k=n entonces
      éxito:= cierto;
      si no
      fábricasVa1(sol, k+1, usadaMáq, usadoTéc, éxito);
      fsi
      usadaMáq[m]:= falso;
      usadoTéc[t]:= falso     {desmarcar}
    fsi
    t:= t+1;
  fmientras
  m:= m+1;
fmientras
fproc
```

La función que realiza el test de factibilidad es la siguiente:

```
fun aceptable?(f, m, t: 1..n, usadaMáq[1..n], usadoTéc[1..n] de bool)
dev respuesta: bool

respuesta:= ⌐usadaMáq[m] ∧ ⌐usadoTéc[t] ∧ apropiada?(m, f) ∧ conoce?(t,m) ∧ aceptaría?(t,f)
ffun
```

Y la función principal que devuelve la primera solución (si existe) es:

```
fun fábricas(n: nat+) dev ⟨éxito: bool, sol[1..n]
de pareja⟩
var usadaMáq[1..n], usadoTéc[1..n] de bool

usadaMáq[1..n] := [falso];
usadoTéc[1..n] := [falso];
éxito:= falso;
fábricasVa1 (sol, 1, usadaMáq, usadoTéc, éxito);
ffun
```

Otra posibilidad es aplicar parte del test de factibilidad a las máquinas, y no tomar decisiones sobre los técnicos cuando ya no se pasa el test, por lo que la poda es más eficiente. Ello nos conduce al siguiente algoritmo:

```
proc fábricasVa2(sol[1..n] de pareja, e k:
1..n, usadaMaq[1..n], usadoTéc[1..n] de
bool, éxito: bool)

m := 1
mientras m≤n ∧ ⌐éxito hacer
  si ⌐usadaMáq[m] ∧ apropiada?(m, k)
  entonces
    t := 1
    mientras t≤n ∧ ⌐éxito hacer
      si ⌐usadoTéc[t] ∧ conoce?(t, m) ∧
      aceptaría?(t, k) entonces
        sol[k].máquina:= m;
        sol[k].técnico:= t
        usadaMáq[m]:= cierto;
        usadoTéc[t]:= cierto;      {marcar}
        si k=n entonces
          éxito:= cierto;
        si no
          fábricasVa2(sol, k+1, usadaMáq, usadoTéc,
          éxito);
        fsi
        usadaMáq[m]:= falso;
        usadoTéc[t]:= falso      {desmarcar}
      fsi
      t:= t+1;
    fmientras
  fsi
```

```
      m:= m+1;
   fmientras
fproc
```

La función principal no cambia, excepto por el nombre del algoritmo recursivo de vuelta atrás al que se llama.

14.6. Cubos con letras

Tenemos *n* cubos identificados por un número del 1 al *n.* Cada cubo tiene impresa en cada una de sus caras una letra distinta.

Se indica además una palabra de *n* letras.

Se trata de colocar los *n* cubos uno a continuación de otro, de forma que con esa disposición se pueda formar la palabra dada.

(Como en diferentes cubos puede haber letras repetidas, la solución puede no ser única, o no existir).

--------------------------Solución-----------------------

Los cubos vienen dados en un vector C[1..n] donde se representa el cubo C[i] como un vector C[i][1..6] que denota las letras en las 6 caras del cubo. La palabra viene dada en un vector P[1..n] de letras.

Vamos a representar la solución como una tupla $(x_1,\ldots,x_n)$ de pares $\langle i,j\rangle$ indicando que se considera la letra en la cara *j* del cubo C[i]. Se tiene que cumplir que en cada posición se utiliza una cara válida de alguno de los cubos, y que los cubos no se repiten y se ha formado la palabra correcta. El árbol es similar al de la Figura 14.5. donde cada nodo tiene ahora 6*n* hijos.

Como la solución debe ser una permutación de los *n* cubos, llevamos como marcador un vector *usado*[1..n] para saber los cubos que ya han sido colocados. Escribimos una versión que encuentra la primera solución, si es que

existe.

```
tipos
    cubo = vector [1..6] de car
    par = reg
            cubo: 1..n
            cara: 1..6
          freg
ftipos
```

```
proc cubosVa(e P[1..n] de car, e C[1..n]
de cubo, sol[l..n] de par, e k: 1 ,.n,
usado[1..n] de bool, éxito: bool)

cubo:= 1;     {recorre los cubos}
mientras cubo≤n ∧ ⌐éxito hacer
    si ⌐usado[cubo] entonces
        usado[cubo]:= cierto;    {marcar}
        cara:= 1;    {recorre las caras del cubo}
        mientras cara ≤ 6 ∧ ⌐éxito hacer
            si P[k]= C[cubo][cara] entonces
                sol[k].cubo:= cubo;
                sol[k].cara := cara;
                si k=n entonces
                    éxito:= cierto;
                si no
                    cubosVa(P, C, sol, k+1, usado, éxito);
                fsi
            fsi
            cara:= cara + 1;
        fmientras
        usado[cubo]:= falso;    {desmarcar}
    fsi
    cubo:= cubo + 1;
fmientras
fproc
```

La función principal es la siguiente:

```
fun cubos(P[1..n] de car, C[1..n] de
cubo) dev ⟨éxito: bool, sol[1..n] de par⟩
var usado[1..n]

éxito:= falso;
```

```
usado[1..n]:= [falso];
cubosVa(P, C, sol, 1, usado, éxito);
ffun
```

14.7. Pintar grafos

Dados un grafo G y un número $m>0$, queremos descubrir todas las formas de pintar los vértices de G utilizando un máximo de *m* colores, de tal forma que ningún par de vértices adyacentes tengan el mismo color.

----------------------------Solución:-------------------------

Si numeramos los colores de 1 a *m* y el grafo tiene *n* vértices numerados de 1 a *n*, entonces una solución al problema puede representarse como una tupla $(x_1,\ldots,x_n)$. donde $x_i \in \{1..m\}$ es el color con el que se ha pintado el vértice *i*. Las soluciones tienen que cumplir que solo utilizan colores válidos y que 2 vértices adyacentes no tienen el mismo color.

El árbol de exploración es similar al de la Figura 14.1, con *m* hijos por nodo y *n* niveles.

Comprobaremos en cada etapa que el color asignado al vértice *k* no ha sido asignado a ningún vértice adyacente a *k* previamente coloreado. Utilizar en este caso el mecanismo de marcaje no sería sencillo. Tendríamos que mantener, para cada vértice *i* y color *j*, la información de si hay algún vértice adyacente a *i* pintado de color *j*. Guardando dicha información en una matriz *usado[i, j]*, comprobar si podemos pintar de color *j* el vértice *i* tendría un coste constante. Sin embargo, marcar sería costoso ya que tendríamos que recorrer los vértices *k* adyacentes a *i* para marcar *usado[k,* j]. Bastante más complicada resultaría la labor de

desmarcar. Por todo ello, en este caso no utilizamos marcadores.

El algoritmo es el siguiente:

```
proc coloreadoGrafo(e G: grafo[n], e m: nat+, sol[1..n] de 1..m, e k: 1..n)

para color=1 hasta m hacer
  sol[k]:= color;
  si testColor(G, sol, k) entonces
    si k=n entonces
      imprimir(sol);
    si no
      coloreadoGrafo(G, m, sol, k+1);
  fsi
  fsi
fpara
fproc
```

La función testColor, que comprueba que sigue sin haber dos vértices adyacentes ya coloreados con el mismo color, se muestra a continuación. Naturalmente, como al nivel *k* se llega después de comprobar la factibilidad en el nivel *k*-1, sabemos que la tupla $(x_1,\ldots,x_{k-1})$ es factible (no hay vértices adyacentes con el mismo color), por lo que basta comprobar que la propiedad se mantiene con respecto al nuevo vértice x_k.

```
fun testColor(G: grafo[n], sol[1..n] de
1..m, k: 1..n) dev respuesta : bool

respuesta:= cierto;
j:= 1;
mientras j<k ∧ respuesta hacer
   si estáArista?(j, k, G) ∨ estáArista?(k, j,
   G) entonces
      respuesta:= sol[j] ≠ sol[k];
   fsi
   j:= j+1;
fmientras
ffun
```

La llamada inicial para resolver el problema es coloreadoGrafo(G, *m*, *sol*, 1). Nótese que este algoritmo es válido para grafos tanto dirigidos como no dirigidos.

Existen diversas representaciones posibles para el grafo. De las dos que hemos dado en el Capítulo 9 en este caso resulta más apropiada la que utiliza una matriz de adyacencia (véase el Ejercicio 9.3) porque garantiza un coste constante para la operación estáArista?.

14.8. n Reinas

Diseñar un algoritmo que encuentre todas las formas de colocar *n* **REINAS** sobre un tablero de ajedrez de tamaño *n* x n de tal forma que no haya 2 reinas dándose *jaque,* es decir, que no haya 2 reinas en una misma fila, columna o diagonal.

-----------------------------Solución----------------------

Si numeramos las filas y las columnas de 1 a *n* y lo mismo hacemos con las reinas, como cada reina ha de estar en una fila diferente, podemos asumir que la reina *i* estará en alguna posición de la fila *i.* Por tanto, las soluciones pueden representarse por medio de tuplas $(x_1,\dots,x_n)$ donde xi será la columna que ocupa la reina *i* en la fila *i.* Las soluciones tienen que cumplir que cada reina esté colocada en una de las *n* columnas, y que no haya 2 reinas en la misma columna ni en la misma diagonal. La primera condición requiere que cada tupla $(x1,\dots,x_n)$ sea una permutación de $(1,\dots,n\}$.

Falta formular que 2 reinas están sobre una misma diagonal. Para las posiciones sobre una misma diagonal descendente (↘) se cumple que tienen el mismo valor *fila - columna,* mientras que para las posiciones en la misma diagonal ascendente (↗) se cumple que tienen el mismo valor *fila + columna.* Así. si tenemos 2 reinas colocadas en las posiciones $\langle i,j\rangle$ y $\langle k,l\rangle$. entonces

están en la misma diagonal $\Leftrightarrow i-j=k-l \vee i+j=k+l$
$\Leftrightarrow j-l=i-k \vee j-l=k-i$

y por tanto, están en la misma diagonal si y solo si $|j\text{-}l|=|i\text{-}k|$.

En resumen, las restricciones implícitas serán

$$\forall i,j: 1 \leq i,j \leq n: i \neq j$$
$$\Rightarrow \left(x_i \neq x_j \wedge \left|x_i - x_j\right| \neq |i-j|\right).$$

Una vez más, el árbol de exploración es similar al de la Figura 14.1, con *n* niveles.

Al colocar una nueva reina tenemos que comprobar que no dé jaque a ninguna de las reinas ya colocadas (suponiendo que estas no se dan jaque). Veamos cómo podemos utilizar un marcador para evitar un coste lineal en esta comprobación. Hay que comprobar que cada columna y cada diagonal, tanto ascendente como descendente, está ocupada a lo sumo por una reina. En consecuencia, podemos utilizar como marcadores 2 vectores de booleanos que nos digan qué columnas y qué diagonales han sido ya ocupadas. Para ello necesitamos numerar las diagonales. La Figura 14.6 indica cómo podemos hacerlo en el caso *n* = 8. Las diagonales descendentes ↘ se numeran de 1 a 15 y las diagonales ascendentes ↗ de 16 a 30 (las filas se numeran de 1 a 8 de arriba abajo y las columnas también de 1 a 8 de izquierda a derecha).

En el caso general de un tablero de tamaño *n* x *n,* las diagonales descendentes ↘ se numeran de 1 a 2n-1,

y las diagonales ascendentes ↗ de *2n* a 4n-2.

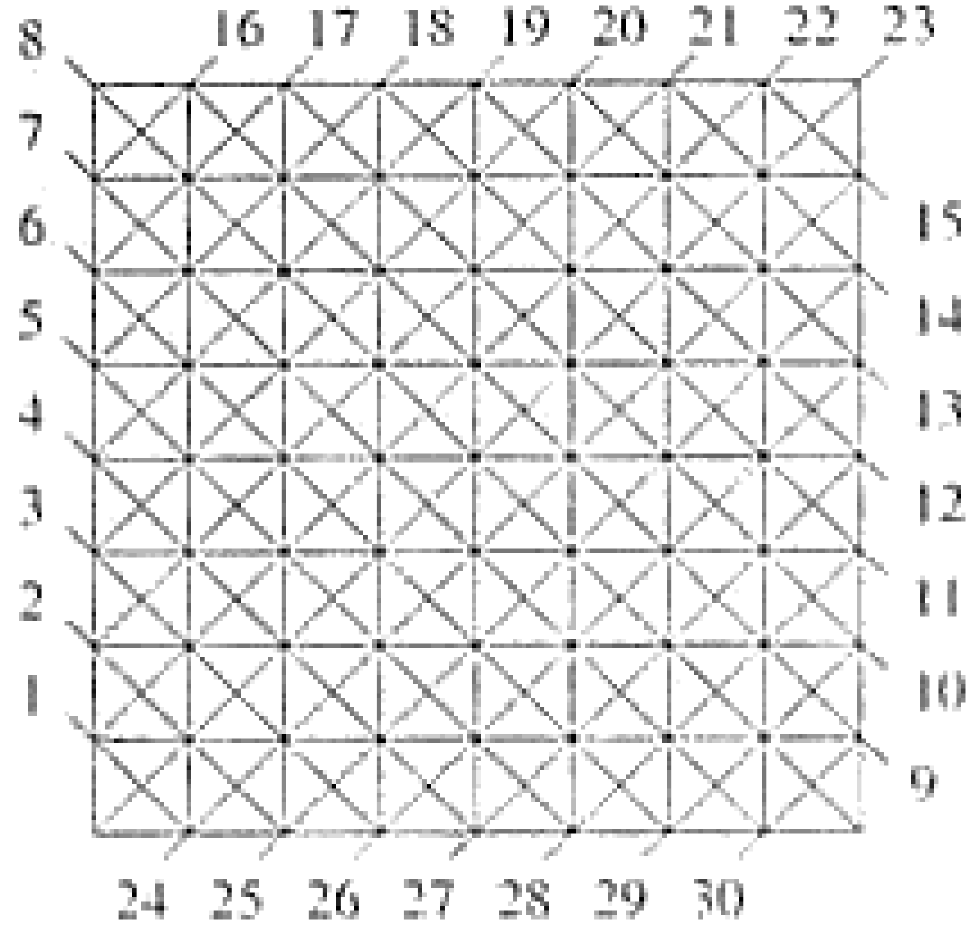


Figura 14.6: Numeración de diagonales de un tablero de tamaño 8x8.

La reina en la posición *⟨i,j⟩* ocupa la diagonal descendente *j-i+n* y la diagonal ascendente *i+j+2n*-2. Con esta información basta consultar 3 posiciones en los 2 vectores de columnas y diagonales ocupadas para obtener el valor del test de factibilidad.

El algoritmo que imprime todas las soluciones queda así:

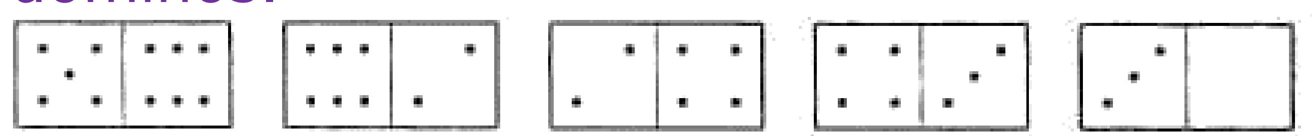
```
proc reinasVa(sol[1..n] de 1..n, e k: 1..n,
C[1..N], D[1..4n-2] de bool)
  para columna = 1 hasta n hacer
    sol[k] := columna;
    si ¬C[sol[k]] ∧ ¬D[sol[k]-k+n] ∧
    ¬D[k+sol[k]+2n-2] entonces
      {marcar}
      C[sol[k]:= cierto;
      D[sol[k]-k+n]:= cierto;
      D[k+sol[k]+2n-2]:= cierto;
      si k=n entonces
        imprimir(sol);
      si no
        reinasVa(sol, k+1, C, D);
      fsi
      {desmarcar}
      C[sol[k]]:= falso;
      D[sol[k]-k+n]:= falso;
      D[k+sol[k]+2n-2]:= falso
    fsi
  fpara
fproc
```

El procedimiento principal que realiza la inicialización adecuada para llamar al procedimiento recursivo es:

```
proc reinas(e n: nat+)
var sol[1..n] de 1 ..n, C[1..n], D[1..4n-2]
de bool
  C[1..n]:= [falso];
  D[1..4n-2]:= [falso];
  reinasVa(sol, 1, C, D);
fproc
```

*14.9. Dominó (Febr 2023)

El juego usual del **DOMINÓ** tiene 28 fichas diferentes. Cada ficha es rectangular y tiene grabado en cada extremo un número de puntos entre 0 y 6. Siguiendo las reglas del juego, las fichas se colocan formando una cadena de tal manera que cada par de fichas consecutivas tienen iguales los números correspondientes a los 2 extremos que se tocan. El siguiente diagrama muestra pane de un ejemplo de cadena de dominós.



Desarrollar un algoritmo que imprima (sin repeticiones) todas las cadenas circulares correctas de 28 fichas.

------------------------Solución-----------------------

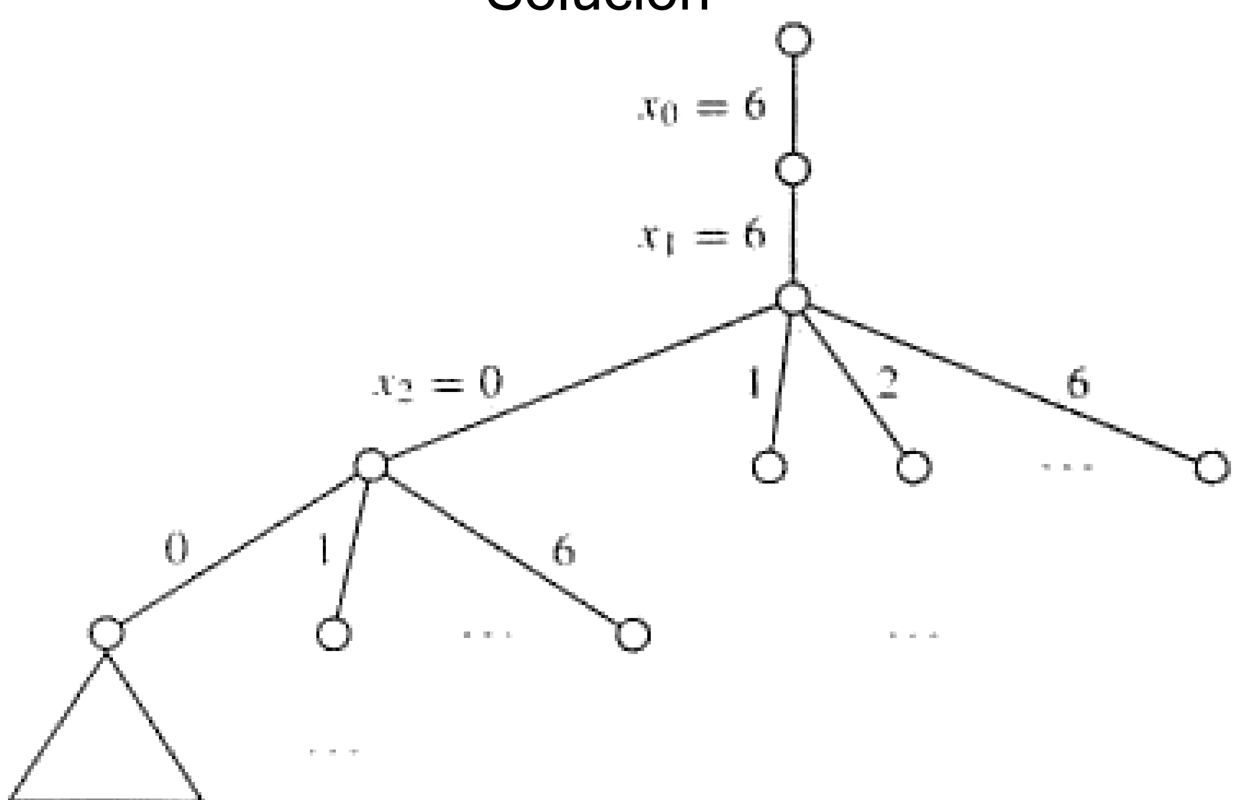


Figura 14.7: Árbol de exploración para el problema del dominó.

La solución se representa como una tupla (x_0,…,x_{28}) de números en el rango 0..6, de forma que implícitamente ya se satisface la restricción de que 2 fichas consecutivas en la

cadena tienen que tener el mismo número en los 2 extremos que se tocan. Por ejemplo, la cadena del enunciado se representa como (5, 6, 2, 4, 3, 0, 1); en esta situación ya sabemos que la siguiente ficha tiene que tener un 1 en un extremo, por lo que solo hay que tomar decisiones con respecto al posible número en el otro extremo.

Como las cadenas son circulares y hay que evitar las repeticiones, vamos a fijar el principio de cada cadena de forma arbitraria en el 6 doble como se hace en el juego habitual de dominó. La forma del árbol de exploración es la mostrada en la Figura 14.7.

Para conocer las fichas usadas, utilizaremos como marcador una matriz (simétrica) *usada*[0..6, 0..6]. Para que las cadenas sean circulares, cuando se han colocado todas las fichas, habría que comprobar que la cadena efectivamente se cierra bien, y así lo hacemos en el siguiente algoritmo, por generalidad; sin embargo, al utilizar las 28 fichas del juego no sería necesario hacer tal comprobación, pues siempre se cierra bien la cadena.

```
proc dominóVa(sol[0..28] de 0..6, e k: 2..28, usada[0..6, 0..6] de bool)
  para j=0 hasta 6 hacer
    si ¬ usada[sol[k-1], j] entonces
      sol[k]:= j;
      {marcar}
      usada[sol[k-1],j]:= cierto ;
      usada[j, sol[k-1]]:= cierto;
      si k=28 entonces
        si sol[28] = sol[0] entonces
          imprimir(sol);
        fsi
      si no
        dominóVa(sol, k+1, usada)
```

```
      fsi
    {desmarcar}
    usada[sol[k-1], j]:= falso;
    usada[j, sol[k-1]]:= falso;
    fsi
  fpara
fproc
```

El algoritmo principal que realiza la inicialización y llamada inicial adecuadas es:

```
proc dominó()
var sol[0..28] de 0..6, usada[0..6, 0..6] de
bool
  sol[0]:= 6;
  sol[1]:= 6;
  usada[0..6, 0..6]:= [falso];
  usada[6,6]:= cierto;
  dominóVa(sol, 2, usada);
fproc
```

Nótese que al haber fijado la primera ficha, empezamos a tomar decisiones para *k*=2. y por tanto el acceso en el algoritmo dominóVa a *sol[k*-1] siempre es correcto.

ChatGPT

El pseudocódigo que proporcionas describe un algoritmo para generar todas las posibles cadenas circulares de dominó utilizando las 28 fichas del juego. El juego de dominó tradicional consta de 28 fichas, cada una con dos números entre 0 y 6. Las cadenas circulares son aquellas donde el último número de la última ficha coincide con el primer número de la primera ficha, formando un círculo.

EXPLICACIÓN DEL ALGORITMO

1. **Inicialización**: Se fija la primera ficha como el 6 doble (6,6). Esto se hace para evitar repeticiones, ya que cualquier cadena circular se puede iniciar con cualquier ficha.

2. **Matriz `usada`**: Esta matriz es una tabla de 7x7 que lleva un registro de qué fichas ya se han usado. Si `usada[i][j]` es verdadero, significa que la ficha con los números i y j ya se ha utilizado.

3. **Función Recursiva (`dominó-va`)**:
   - **Parámetros**:
     - `sol`: Un arreglo que representa la cadena actual de fichas.
     - `k`: El índice actual en la cadena de fichas.
     - `usada`: La matriz que indica qué fichas se han usado.
   - **Proceso**:
     - Se intenta colocar una ficha en la posición `k` de la cadena, asegurándose de que el número en un extremo de la nueva ficha coincida con el número en el extremo correspondiente de la ficha anterior (`sol[k-1]`).
     - Si una ficha es colocada, se actualiza la matriz `usada` para marcar esa ficha como utilizada.
     - Si se alcanza la ficha 28 y esta cierra correctamente la cadena (es decir, su segundo número coincide con el primer número de la ficha inicial), se imprime la solución.
     - Si no se ha terminado, se llama a la función recursivamente para el siguiente índice `k+1`.

CONVERSIÓN A JAVA

El pseudocódigo se puede convertir a Java de la siguiente manera:

```java
public class DominoCirculares {
    public static void main(String[] args) {
        int[] sol = new int[29]     // Java indexa desde 0, por lo que hay 29 posiciones
        boolean[][] usada = new boolean[7][7]   // Matriz de 7x7
        sol[0] = 6
        sol[1] = 6
        usada[6][6] = true
        dominoVa(sol, 2, usada)
    }

    private static void dominoVa(int[] sol, int k, boolean[][]
```

```
usada) {
        if (k == 28) {
            if (sol[28] == sol[0]) {
                imprimir(sol)
            }
            return;
        }

        for (int j = 0; j <= 6; j++) {
            if (!usada[sol[k - 1]][j]) {
                sol[k] = j
                usada[sol[k - 1]][j] = true
                usada[j][sol[k - 1]] = true
                dominoVa(sol, k + 1, usada)
                usada[sol[k - 1]][j] = false
                usada[j][sol[k - 1]] = false
            }
        }
    }

    private static void imprimir(int[] sol) {
        for (int i : sol) {
            System.out.print(i + " ")
        }
        System.out.println();
    }
}
```

Este código Java implementa la lógica del pseudocódigo proporcionado. La función `dominoVa` es una función recursiva que construye la cadena de fichas. Cuando se encuentra una cadena válida, la función `imprimir` muestra esta cadena.

14.10. Caballo

Dado un tablero de ajedrez de *n* x n posiciones, y un **CABALLO** colocado en una posición arbitraria ⟨x, y⟩, se pide un algoritmo que encuentre (si existe) una secuencia de n^2 - 1 movimientos de caballo de forma que se visiten todos los cuadros del tablero una sola vez.

Modificar el algoritmo añadiendo el requisito adicional de que, con un último movimiento, el caballo debe volver a la casilla de partida.

---------------------------Solución---------------------

Podemos dar la solución en una tupla de tamaño fijo con n^2 componentes donde cada componente es una posición del tablero. Si la tupla es (x_1,..., x_{n^2}), xi será la i-ésima posición del tablero por donde pasa el caballo. Además se debe cumplir que el caballo no pase dos veces por una misma posición del tablero (la tupla es una permutación de todas las casillas) y que el caballo realice un movimiento lícito para pasar de una casilla a la siguiente. En general el caballo podrá realizar 8 movimientos, por lo que el árbol de exploración es similar al de la Figura 14.1, donde cada nodo tiene 8 hijos y hay n^2 niveles.

Utilizaremos un marcador para conocer las casillas del tablero por las que el caballo ya ha pasado:

usada[i, j] ⇔ el caballo ha pasado por la casilla ⟨*i, j*⟩.

El algoritmo que encuentra la primera solución es:

```
tipos
    posición = reg
            fila: 1..n
            columna: 1..n
        freg
ftipos
{el caballo está en la casilla sol[k - 1]}
proc caballoVa(sol[1 ..n²] de posición, e k
:1..n², usada[1..n, 1..n] de bool, éxito: bool)

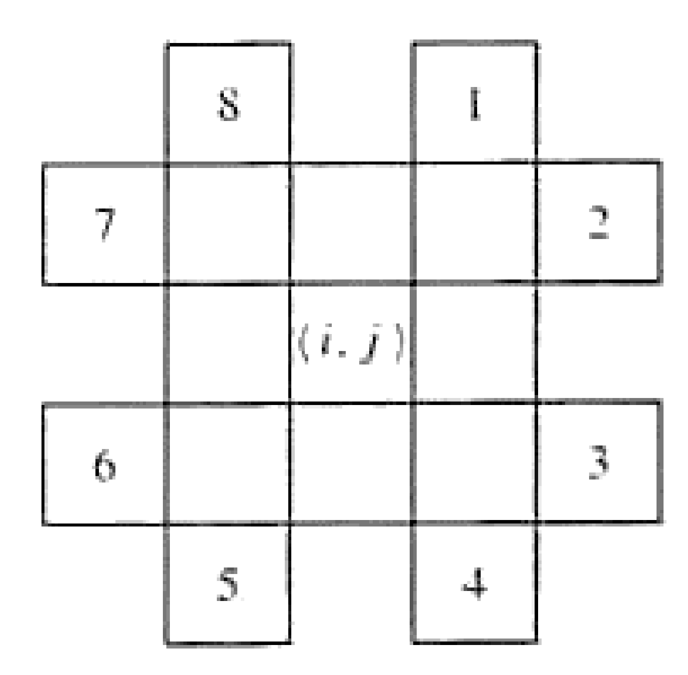
 movimiento:= 1;  {genera movimientos}
 mientras movimiento≤ 8 ∧ ¬éxito hacer
   sol[k] := nuevaCasilla(sol[k-1], movimiento);
   ⟨i,j⟩ :=⟨sol[k].fila, sol[k].columna⟩;
   si 1≤i ∧ i≤n ∧ 1≤j ∧ j≤n ∧ ¬usada[i, j]
   entonces
     usada[i,j]:= cierto;     {marcar}
     si k=n² entonces
       éxito:= cierto;
     si no
       caballoVa(sol, k+1, usada, éxito);
     fsi
     usada[i,j]:= falso     {desmarcar}
   fsi
   movimiento:= movimiento + 1  {siguiente
   movimiento}
 fmientras
fproc
```

La función nuevaCasilla genera los distintos movimientos que puede hacer un caballo desde una posición dada.

```
fun nuevaCasilla(act: posición, mov : 1..8) dev
sig: posición

 ⟨i, j⟩ := ⟨act.fila, act.columna⟩;
  casos
    mov=1 →⟨i', j'⟩:= ⟨i-2, j+1⟩;
  □ mov=2 → ⟨i', j'⟩:= ⟨i-1, j+2⟩;
  □ mov=3 → ⟨i', j'⟩:= ⟨i+1, j+2⟩;
  □ mov=4 → ⟨i', j'⟩:= ⟨i+2, j+1⟩;
  □ mov=5 → ⟨i', j'⟩:= ⟨i+2, j-1⟩;
```

```
▯ mov=6 → ⟨i', j'⟩:= ⟨i+1, j-2⟩;
▯ mov=7 → ⟨i', j'⟩:= ⟨i-1, j-2⟩;
▯ mov=8 → ⟨i', j'⟩:= ⟨i-2, j-1⟩;
fcasos
⟨sig.fila, sig.columna⟩:= ⟨i', j'⟩;
ffun
```



La función principal, con la correspondiente inicialización, es:

```
fun caballo(n: nat+, pos: posición) dev
⟨éxito: bool, sol[1..n²] de posición⟩
var usada[1..n, 1.n] de bool

sol[1]:= pos;  {desde donde empieza el caballo }
usada[1..n, 1..n] := [falso];
usada[pos.fila, pos.columna]:= cierto;
éxito:= falso;
caballoVa(sol, 2, usada, éxito);
ffun
```

Nótese que al empezar con $k=2$ porque la primera casilla viene dada, el acceso a *sol*[*k*-1] siempre es correcto.

La modificación en la cual se comprueba que el recorrido se cierra volviendo a la casilla inicial es

```
si k=n² entonces
   éxito:= cierra?(sol[n²], sol[1]);
si no caballoVa(sol, k+1, usada, éxito);
fsi
```

donde la función auxiliar cierra?(⟨*i*, *j*⟩, ⟨*i*', *j*'⟩) comprueba que se puede hacer un movimiento de caballo desde la casilla ⟨*i*, *j*⟩ a la ⟨*i*', *j*'⟩.

```
fun cierra?(pos1, pos2: posición) dev respuesta : bool

respuesta:= falso;
movimiento:= 1;
   mientras movimiento ≤ 8 ∧ ¬respuesta hacer
      respuesta:= (pos2 = nuevaCasilla(pos1, movimiento));
      movimiento:= movimiento + 1;
   fmientras
ffun
```

14.11. Dividir botín

El Maqui y *el Popeye* acaban de desvalijar la reserva nacional de oro. Los lingotes están empaquetados en *n* cajas de diferentes pesos (reales) y, como no tienen tiempo de desempaquetarlos para dividir el botín, deciden basarse en los pesos de las cajas para intentar distribuir el botín a medias. Al cabo de un buen rato todavía no han conseguido repartirse el botín, por lo cual acuden al *Teclas* para saber si el botín se puede dividir en 2 partes iguales sin desempaquetar ninguna de las cajas

------------------------Solución-------------------

Este problema fue resuelto mediante el método de *programación dinámica* en el Ejercicio 13.4, con la restricción de que los pesos tenían que ser números naturales. Veamos ahora una solución mediante *vuelta atrás* del caso más general en el que los pesos son reales.

Ya vimos en el Ejercicio 13.4 que el problema es equivalente a comprobar si es posible, sumando algunos de los pesos, obtener *P/2* donde $P = \sum_{i=1}^{n} p_i$.

Veamos cómo buscar un subconjunto de los *pi* cuya suma sea igual a cierta cantidad dada. C. Supondremos que $\sum_{i=1}^{n} p_i \geq C$ ya que en otro caso no hay solución, y que ∀i: $1 \leq i \leq n : p_i \leq C$ para no considerar pesos que sabemos de antemano no tiene sentido sumar.

La solución puede darse mediante una tupla $(x_1,\ldots,x_n)$ que represente la función característica de este subconjunto, es decir, $x_i=1$ si el elemento p_i pertenece al subconjunto y $x_i=0$ en caso contrario. El árbol

de exploración, con *n* niveles, tiene la forma de la Figura 14.8.

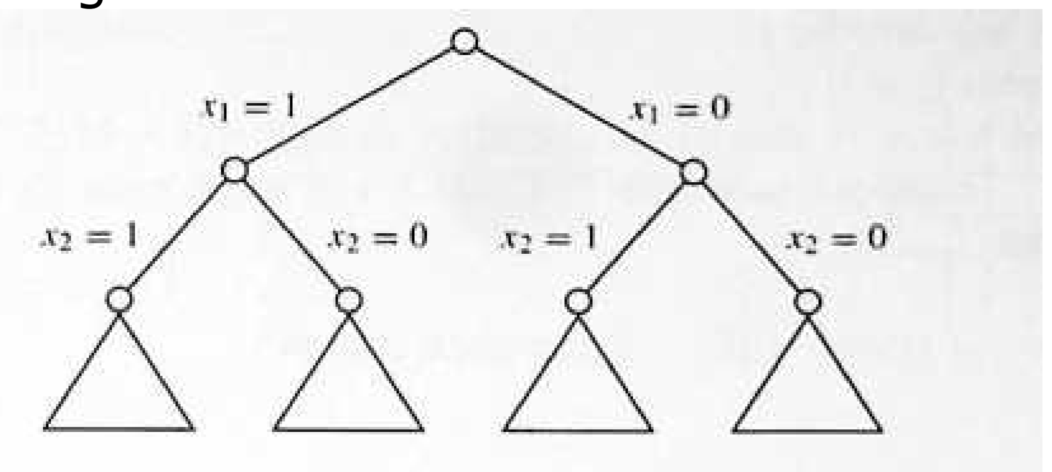


Figura 14.8: Árbol de exploración para el problema del reparto.

Probaremos primero con x_k=1 pues es plausible, aunque no seguro, que al reducir el valor de la suma pendiente de acumular facilitemos el acceso más rápido a una solución. Llevamos como marcador una variable *suma* que representa la suma acumulada de los números elegidos hasta el momento. Así. cuando *suma* sea igual a *C* podemos acabar (los restantes pesos no considerados no se cogen). El algoritmo recursivo de vuelta atrás es el siguiente:

```
proc conseguirCantidad1(e P[1..n] de
real+, e C: real, sol[1..n] de 0..1, e k: 1..n,
suma: real, éxito: bool)

{hijo izquierdo — añadir peso}
si suma + P[k] ≤ C entonces
    sol[k]:= 1;
    suma:= suma + P[k];     {marcar}
    si suma = C entonces
        éxito:= cierto;
        sol[k + 1..n]:= [0];
    si no
        si k<n ∧ suma< C entonces
            conseguirCantidad1 (P, C, sol, k+1,
            suma, éxito);
        fsi
    fsi
    suma:= suma - P[k];     {desmarcar}
```

```
fsi

{hijo derecho — no añadir peso, no puede generar solución}
si ¬éxito ∧ k<n entonces
    sol[k]:= 0;
    conseguirCantidad1(P, C, sol, k+1, suma, éxito);
fsi
fproc
```

Podemos realizar una mayor poda, mejorando la eficiencia del algoritmo, si aprovechamos el hecho de que los pesos son positivos. En ese caso, si *suma* + $\sum_{i=k+1}^{n} P[i]$ < C, podemos podar el nodo actual porque nunca vamos a poder alcanzar la solución desde él. Además, si los pesos están ordenados de forma creciente, también podemos podar el nodo actual si *suma* + *P[k*+1] > *C.* Para comprobar fácilmente la primera condición llevamos otro marcador *resto* tal que *resto* = $\sum_{i=k+1}^{n} P[i]$.

En el algoritmo, al probar el hijo izquierdo no es necesario comprobar la primera condición, ya que si era posible alcanzar *C* para el padre y hemos cogido un nuevo peso, sigue siendo posible.

{*resto* = $\sum_{i=k}^{n} P[i]$ ∧ P[1] ≤ … ≤ P[n]}

```
proc conseguirCantidad2(e P[1..n] de real⁺, e C: real, sol[1..n] de 0..1. e k: 1..n, suma, resto: real, éxito: bool)
```

```
resto:= resto - P[k]; {marcar común a los 2 hijos}
{hijo izquierdo}
sol[k]:= 1;
suma:= suma + P [k];   {marcar}
si suma = C entonces
   éxito:= cierto;
   sol[k + 1..n]:= [0];
si no
   si k<n ∧ suma + P[k+1] < C entonces
      conseguirCantidad2(P, C, sol, k+1, suma, resto,
      éxito);
   fsi
fsi

suma:= suma - P[k];    {desmarcar}
si ¬éxito entonces
   {hijo derecho}
   sol[k] := 0;
   si k<n ∧ suma + resto ≥ C ∧ suma + P[k+1]≤ C
   entonces
      conseguirCantidad2(P, C, sol, k+1, suma, resto,
      éxito);
   fsi
fsi
resto:= resto + P[k];  {desmarcar común}
fproc
```

La función principal que comprueba si se puede repartir el botín en 2 partes iguales es:

```
fun repartirBotínVa(P[1..n] de real⁺) dev
⟨éxito: bool, sol[1..n] de 0..1 ⟩

peso:= 0;
para i=1 hasta n hacer
   peso:= peso + P[i];
fpara      {peso = ∑_{i=k}^{n} P[i]}
suma:= 0;
resto:= peso;
éxito:= falso;
conseguirCantidad2(P, peso/2, sol, 1, suma,
resto, éxito);
ffun
```

14.12. Laberinto booleano

A partir de una matriz booleana L[1..n, 1..n] se puede representar un LABERINTO de la siguiente forma: a partir de una casilla dada, los movimientos posibles son desplazarse a cada una de las 4 casillas adyacentes (vertical y horizontalmente). Si L[i,j] = cierto se puede pasar por la casilla ⟨*i, j*⟩, y si L[i, j] = falso no se puede pasar por esa casilla.

Suponiendo que L[1,1] = *L*[*n, n*] = cierto, escribir un algoritmo que encuentre, si existe, un camino de la casilla ⟨1,1⟩ a la ⟨n,n⟩.

--------------------------Solución-----------------------

Las soluciones son listas de posiciones del laberinto de longitud n^2 como máximo (que representaremos mediante un vector *sol*[1..n^2]), tales que las posiciones son válidas, están libres y cada una es adyacente con la siguiente. En principio, desde cada casilla podemos realizar 4 movimientos, por lo que el árbol de exploración es similar al de la Figura 14.1, donde cada nodo tiene 4 hijos y hay n^2 niveles como máximo.

Para controlar que no pasamos dos veces por el mismo sitio, mantendremos una matriz *usada*[*i*,j] que indica si hemos pasado ya por la casilla ⟨*i, j*⟩.

Utilizamos 2 funciones auxiliares: casillaValida? y siguiente.

La función casillaVálida? comprueba si cierta posición es una posición correcta del tablero, no visitada y por la que

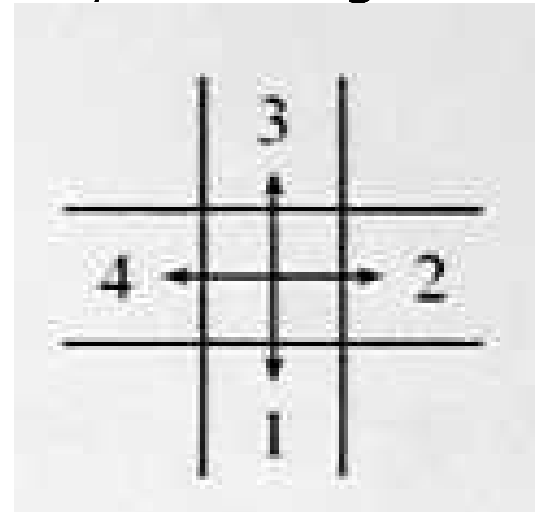
podemos pasar.

```
tipos
      posición = reg
            fila: 1..n
            columna: 1..n
           freg
ftipos
```

```
fun casillaVálida?(L[1..n, 1..n],
usada[1..n, 1..n] de bool, pos: posición)
dev respuesta: bool

⟨i, j⟩:= ⟨pos.fila, pos.columna⟩;
respuesta:= 1≤i ∧ i≤n ∧ 1≤j ∧ j≤n ∧ L[i, j] ∧
¬usada[i,j];
ffun
```

La función siguiente devuelve la casilla a la que hay que pasar según el número de movimiento, de la siguiente manera:




```
fun siguiente(m: 1..4, act: posición) dev
sig: posición

⟨i, j⟩:= ⟨act.fila, act.columna⟩
casos
  m=1 → ⟨i', j'⟩:= ⟨i+1, j⟩
 m=2 → ⟨i', j'⟩:=⟨i, j+1⟩
 m=3 → ⟨i', j'⟩:= ⟨i-1, j⟩
 m=4 → ⟨i', j'⟩:= ⟨i,j-1⟩
fcasos
⟨sig.fila, sig.columna⟩:= ⟨i', j'⟩
ffun
```

Como las soluciones son de longitud variable, el algoritmo devuelve además del vector *sol*, qué posiciones de *sol* son las utilizadas, *númMovimientos*.

El algoritmo que busca la primera solución es el siguiente:

```
proc laberintoVa(e L[1..n, 1..n] de bool,
sol[1..n²] de posición, e k: 1..n², usada[1
..n, 1..n] de bool, éxito: bool,
númMovimientos: 1..n²)

m:=1;    {genera movimientos}
mientras m≤4 ∧ ¬éxito hacer
  pos:= siguiente(m, sol[k-1])
  si casillaVálida?(L, usada, pos) entonces
    sol[k]:= pos;
    usada[pos.fila, pos.columna]:= cierto
    {marcar}
    si pos.fila=n ∧ pos.columna=n entonces
      éxito:= cierto;
      númMovimientos:= k;
    si no
      laberintoVa(L, sol, k+1, usada, éxito,
      númMovimientos)
    fsi
  fsi
  m:= m+1 {siguiente movimiento}
fmientras
fproc
```

El algoritmo *no desmarca* las posiciones utilizadas ya que al volver de la llamada recursiva, o hemos encontrado la solución y hemos acabado, o si no. la última posición por la que hemos pasado no nos ha llevado a la salida, por lo que no nos interesa volver a considerarla. Dejándola marcada, las sucesivas llamadas recursivas no la considerarán.

La función principal que devuelve si

existe solución y. en ese caso, por cuáles casillas hay que pasar, es la siguiente.

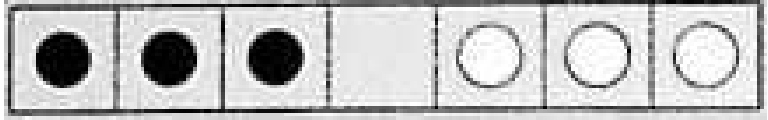
```
fun laberinto(L[1..n, 1..n] de bool) dev
⟨éxito: bool, sol[1..n²] de posición,
númMovimientos: 1..n²⟩
var usada[1..n, 1..n] de bool

⟨sol[1].fila, sol[1].columna⟩ := ⟨1, 1⟩;
usada[1..n, 1..n]:= [falso];
usada[1, 1]:= cierto;
éxito:= falso;
laberintoVa(L, sol, 2, usada, éxito,
númMovimientos)
ffun
```

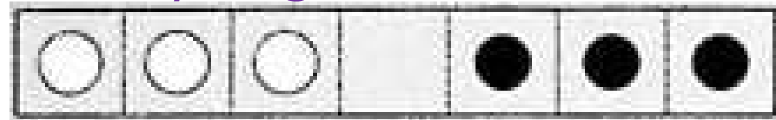
Ya que la primera llamada se hace con *k*=2, siempre es correcto el acceso a *sol*[k-1] en el algoritmo laberintoVa.

14.13. Mover fichas

Consideremos un tablero formado por 7 casillas adyacentes, con 3 fichas blancas y 3 fichas negras colocadas de la siguiente forma.



El objetivo del juego consiste en llegar al siguiente estado, donde la situación de las fichas blancas y negras se ha intercambiado.



Cada ficha solo puede moverse en el sentido correspondiente a su color, o sea, una ficha blanca de derecha a izquierda y una negra de izquierda a derecha (según los dibujos anteriores), pero los únicos movimientos posibles son:

- avanzar si la casilla de delante está vacía,
- saltar una ficha que se tenga delante (el color no importa), si esta tiene a su vez un hueco vacío delante.

Generalizando la situación a un tablero de tamaño $n = 2f + 1$ ocupado inicialmente por f fichas negras a la izquierda y otras tantas blancas a la derecha, escribir un algoritmo que encuentre una secuencia de movimientos para pasar del estado inicial al final.

------------------------Solución---------------------------

Un estado del tablero se representa mediante un vector T[1..n] de números naturales en el rango 0..2, donde 0 representa la casilla *vacía*, **1** una ficha de color negro y **2** una ficha de color **blanco**. Nótese además que $f = n$ div 2.

Los movimientos del juego se reflejan en el estado del tablero por el desplazamiento de la casilla vacía. Según el enunciado, en general en cada estado hay 4 posibilidades para mover la casilla vacía, según una ficha blanca

avance una o 2 posiciones a la izquierda, o una ficha negra avance una o 2 posiciones a la derecha: sin embargo, hay estados en los que algunos de estos movimientos pueden no ser posibles, por ejemplo cuando la casilla vacía está en un extremo del tablero.

La función "mover" considera la casilla vacía en la posición *i* del tablero *y* enumera los 4 posibles movimientos indicando con un 0 que el movimiento no es posible, y con un número mayor que 0 la posición de la ficha que se mueve al hueco.

```
fun mover(T[1..n] de 0..n, i:1..n, j:1..4))
dev mov: 0..n

mov:= 0
casos
  j=1 → si i>1 ∧ T[i-1]=1 entonces mov:= i-1
fsi
 j=2 → si i>2 ∧ T[i-2]=1 entonces mov:= i-2
fsi
 j=3 → si i<n ∧ T[i+1]=2 entonces mov:=
i+1 fsi
 j=4 →si i<n-1 ∧ T[i+2]=2 entonces mov:=
i+2 fsi
fcasos
ffun
```

Las soluciones pueden representarse mediante listas de posiciones, representando la ficha que se mueve a la casilla vacía en cada paso. Ya que las fichas solo pueden avanzar en el sentido correspondiente, es fácil concluir que el número total de movimientos está acotado superiormente por n^2, por lo que representaremos las soluciones mediante un vector *sol*[1..n^2]. El algoritmo devolverá además el número de movimientos necesarios, por lo que en realidad la solución vendrá dada en *sol*[1..*numMovimientos]*. El árbol de exploración es similar al de la Figura

14.1, con 4 hijos por nodo, que representan las 4 fichas que pueden moverse, y una profundidad n^2 como máximo.

Para conocer en cada momento cuál es el estado del juego, utilizaremos como marcadores un vector *T*[1..n] que representa el estado del tablero, y un natural *vacia* que representa la posición de la casilla vacía.

El algoritmo que encuentra la primera solución es el siguiente:

```
proc juegoVa(sol[1..n²] de 1..n, e k: 1..n², T[1..n] de 0..2, vacía: 1..n, éxito: bool, númMovimientos: 1..n²)

j:= 1;
mientras j≤4 ∧ ¬éxito hacer
   mov := mover(T, vacia, j);
   si mov ≠ 0 entonces
      sol[k] := mov;
      intercambiarVacía(T, vacia, mov); {marcar}
      si estadoFinal?(T) entonces
         éxito := cierto;
         numMovimientos := k;
      si no
         juegoVa(sol, k+1, T, vacía, éxito, númMovimientos);
      fsi
      intercambiarVacía(T, vacía, mov); {desmarcar}
   fsi
   j = j+1; {siguiente movimiento}
fmientras
fproc
```

La función intercambiarVacia(*T, i, j*) intercambia en el vector *T* los valores en las posiciones *i* y *j* y además también intercambia los valores de estas variables para seguir el rastro de la casilla vacía para que después se pueda desmarcar.

```
proc intercambiarVacia(T[1..n] de 0..2, i, j: 1..n)
⟨T[i], T[j]⟩ := ⟨T[j], T[i]⟩;
⟨i,j⟩ := ⟨j,i⟩;
fproc
```

La función auxiliar que comprueba si un estado dado coincide con el final es la siguiente:

```
fun estadoFinal?(T[1..n] de 0..2) dev respuesta: bool
c := 1;
f := n div 2;
respuesta := cierto
mientras respuesta ∧ c ≤ f hacer
    respuesta := (T[c] = 2);
    c := c+1
fmientras
respuesta := respuesta ∧ (T[f+1] = 0)
c := f + 2;
mientras respuesta ∧ c ≤ n hacer
    respuesta := (T[c] = 1);
    c := c + 1;
fmientras
ffun
```

En realidad, el último bucle es innecesario si suponemos que comenzamos con un tablero correcto y los movimientos solo alteran las posiciones de las fichas.

La función principal que pone en el tablero el estado inicial del juego, busca la primera solución y la devuelve (si existe) es:

{*n* impar}

```
fun juego(n: nat⁺) dev ⟨éxito: bool, sol[1..n²] de 1..n, númMovimientos: 1..n²⟩
var T[1..n] de 0..2

f := n div 2;
T[1..f] := [1];
T[f+1] = 0;
T[f + 2..n] := [2];
vacía := f+1;
éxito := falso;
juegoVa(sol, 1, T, vacía, éxito, númMovimientos);
ffun
```

14.14. Circuitos tren

Un vendedor ambulante tiene que recorrer *n* ciudades, volviendo tras ello al punto de partida. El vendedor conoce las posibles conexiones directas por ferrocarril entre las ciudades, y las tarifas correspondientes. El vendedor desea conocer:

a) todos los circuitos en tren que recorran cada ciudad exactamente una vez y regresen a la ciudad de partida:
b) un circuito en tren que recorra cada ciudad exactamente una vez y regrese a la ciudad de partida, y cuya tarifa total sea mínima.

----------------------------Solución------------------------

Podemos representar la información recogida por el vendedor mediante un grafo dirigido y valorado donde los vértices son las ciudades (numeradas de 1 a *n) y* las aristas son las conexiones directas de ferrocarril, valoradas con la tarifa correspondiente. El problema consiste entonces en encontrar 2 cosas:

(a) todos los *ciclos hamiltonianos* del grafo (véase la Sección 9.1). y
(b) un ciclo hamiltoniano de coste mínimo.

Apartado (a)------------------

Podemos construir las soluciones como tuplas $(x_1,\ldots,x_n)$. donde x, es el vértice por el que pasamos en í-ésimo lugar. Las soluciones tienen que cumplir que se utilizan vértices válidos, que siempre hay arista de uno al siguiente, y que hay arista del último al primero. Como las

soluciones son ciclos, para evitar soluciones repetidas (véase el Ejercicio 14.9) fijamos el comienzo de los ciclos en el vértice 1. es decir, $x_1=1$, y empezaremos a tomar decisiones a partir del segundo vértice del ciclo.

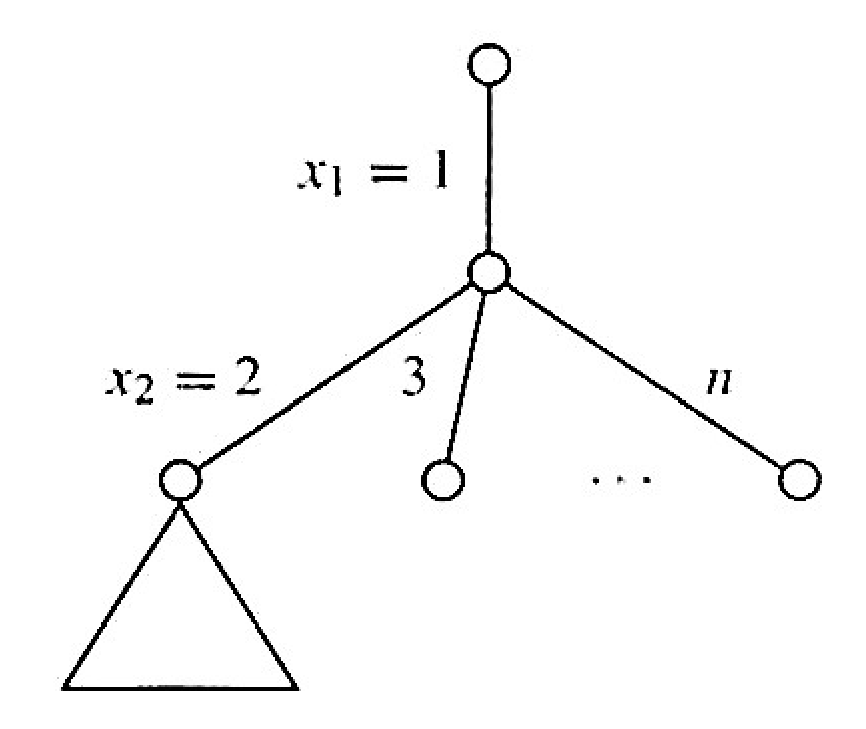


Figura 14.9. Arbol de exploracion para el problema del vendedor

El árbol de exploración es el de la Figura 14.9, donde cada nodo excepto la raíz tiene *n*-1 hijos, y hay *n* niveles.

Las podas se realizan si, al asignar un vértice a una posición de la tupla parcial, por ese vértice ya se ha pasado antes o bien no está conectado al último vértice de la tupla. Mantenemos en un vector *usado*[1..n] de booleanos información sobre los vértices ya incluidos en la solución parcial calculada hasta el momento.

El algoritmo que resuelve el problema de los ciclos hamiltonianos es el siguiente:

```
proc cicloHamiltonianoVa(e G:
grafoVal[n], sol[1..n] de 1..n, e k: 1..n,
usado[1..n] de bool)

para vértice=2 hasta n hacer
  si ⌐usado[vértice] ∧ gvEstáArista?(sol[k-1],
  vértice, G) entonces
    sol[k]:= vértice
    usado[vértice]:= cierto    {marcar}
    si k=n entonces
      {falta comprobar que se cierra el ciclo}
      si gvEstáArista?(sol[n], 1, G) entonces
      imprimir(sol) fsi
    si no
      cicloHamiltonianoVa(G, sol, k+1, usado)
    fsi
    usado[vértice]:= falso   {desmarcar}
  fsi
fpara
fproc
```

El procedimiento principal que hace la inicialización adecuada y llama al anterior es:

```
proc cicloHamiltoniano(e G : grafo-val[n])
var sol[1..n] de 1..n, usado[1..n] de bool

sol[1] := 1
usado[1] := cierto;
usado[2..n] := [falso]
cicloHamiltonianoVa(G, sol, 2, usado)
ffun
```

Ya que se empieza con *k*=2, el acceso a *sol[k-1]* en el algoritmo cicloHamiltonianoVa siempre es correcto.

Aunque hemos utilizado un grafo valorado, en realidad para resolver este problema los valores de las aristas son irrelevantes y el algoritmo es válido también para grafos dirigidos no valorados. La única operación que se

realiza sobre el grafo es determinar si existe una arista, la cual resulta de coste constante cuando el grafo se representa mediante la matriz de valores (véase el Ejercicio 9.6).

Apartado (b)--------------------

Ahora modificamos el algoritmo ciclo-hamiltoniano-va para encontrar el ciclo de coste mínimo según el esquema de optimización (Sección 14.2.3). Necesitamos guardar la mejor solución encontrada hasta el momento, junto con su valor correspondiente: *⟨solMejor, costeMejor⟩*. Mantendremos la mejor solución encontrada hasta el momento, y cada vez que lleguemos a una nueva solución la compararemos con la mejor, actualizando esta si procede. Para evitar cálculos repetidos, mantendremos en un marcador *coste* el coste de la solución parcial, que se podrá ir calculando de forma incremental.

Podemos mejorar la eficiencia podando también nodos de los cuales sabemos que no nos llevarán a ninguna solución mejor que la mejor solución actual *⟨solMejor, costeMejor⟩*. Si para una solución parcial se tiene que *coste* ≥ *costeMejor*, podemos podarla (y no continuar con ella) ya que todas las soluciones en su subárbol tendrán un coste mayor o igual que *coste* (y por tanto que *costeMejor*), puesto que los

valores de las aristas son positivos.

Pero el coste de estas soluciones será de la forma

$$\underbrace{\sum_{i=2}^{k} \text{gv-valor}\ (x_{i-1}, x_i, G)}_{\text{fijo}}$$

$$+\underbrace{\left(\sum_{i=k+1}^{n} \text{gv-valor}\ (x_{i-1}, x_i, G)\right) + \text{gv-valor}\ (x_n, x_1, G)}_{n-k+1 \text{ aristas}}$$

por lo que podemos aproximar el coste de las últimas *n*-*k*+1 aristas con (*n*-*k*+1) *mínG*, donde *mínG* es el valor mínimo de todas las aristas de G. Así se pueden podar aún más nodos no prometedores.

Con estas modificaciones el algoritmo queda así:

```
proc vendedorVa(e G: grafoVal[n], e
mínG: real, sol[1..n] de 1..n, e k: 1..n,
coste: real, usado[1..n] de bool,
solMejor[1..n] de 1..n, costeMejor: real∞)

anterior:= sol[k-1]
para vértice=2 hasta n hacer
 si ⌐usado[vértice] ∧ gvEstáArista?(anterior,
 vértice, G) entonces
  sol[k]:= vértice;
  usado[vértice]:= cierto;    {marcar}
  coste:= coste + gvValor(anterior, sol[k], G);
  si k=n entonces
   si gvEstáArista?(sol[n], 1, G) ∧c
   coste + gvValor(sol[n]. l.G) < costeMejor
   entonces
   solMejor:= sol;
   costeMejor:= coste + gvValor(sol[n, 1, G)
   fsi
  si no
   costeEstimado := coste + (n-k+1) * mínG
   si costeEstimado < costeMejor entonces
   {se puede mejorar solMejor}
```

```
vendedorVa(G, mínG, sol, k+1, coste,
usado, solMejor, costeMejor}
fsi
fsi
usado[vértice]:= falso   {desmarcar}
coste:= coste - gvValor(anterior, sol[k], G)
fsi
fpara
fproc
```

La función principal que hace la inicialización y llama al procedimiento anterior para buscar la mejor solución es la siguiente:

```
fun vendedor(G: grafo-val[n]) dev
(solMejor[1..n] de 1..n, costeMejor : real∞)
var sol[1..n] de 1..n, usado[1..n] de bool

mínG:= cálculoMínimo(G)
sol[1]:= 1
coste:= 0
usado[1]:= cierto;
usado[2..n]:= [falso]
costeMejor +∞
vendedorVa(G, mínG, sol, 2, coste, usado,
solMejor, costeMejor)
ffun
```

Como se empieza con *k*=2, el acceso a *sol*[*k*-1] en el algoritmo vendedorVa siempre es correcto. Si *costeMejor* termina siendo +∞ es porque no hay solución.

La función que calcula el mínimo global es la siguiente:

```
fun cálculo-minimo(G: grafoVal[n]) dev mínG : real∞
mínG := +oo
para i = 1 hasta n hacer
   para j = 1 hasta n hacer
      si i ≠ j ∧ gv-está-arista?(i, j, G) entonces
         mínG:= mín(minG, gv-valor(i, j, G))
      fsi
   fpara
fpara
ffun
```

Las operaciones sobre grafos que se utilizan en estos algoritmo son determinar si existe una arista y devolver su valor. Estas operaciones se pueden realizar con coste constante si el grato se representa mediante la matriz de valores (véase el Ejercicio 9.6).

-14.15. Funcionarios

El Ministro de Desinformación y Decencia se ha propuesto hacer trabajar en firme a sus n funcionarios, para lo que se ha sacado de la manga *n* trabajos. A pesar de su ineficacia, todos los funcionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros y unos los hacen peor que otros. La información al respecto se recoge en 2 tablas T[1..n, 1..n] y E[1..n,1..n] donde T[i,*j*] representa el tiempo que el funcionario *i* tarda en realizar el trabajo *j* y E[i,*j*] representa la eficacia con la que el funcionario *i* realiza el trabajo *j.* Para justificar su puesto, el ministro desea conocer la asignación óptima de trabajos a funcionarios en cada uno de los dos sentidos diferentes (e independientes) siguientes:

a) de modo que la suma total de tiempos sea mínima, y
b) de modo que la suma total de eficacias sea máxima.

----------------------------Solución---------------------

En ambos casos podemos representar la solución en tuplas de la forma $(x_1,...,x_n)$ donde x_i es el trabajo asignado al funcionario *i.* Las soluciones tienen que cumplir que son permutaciones de los n trabajos. El árbol de exploración es similar al de la Figura 14.1. donde cada nodo tiene *n* hijos y hay *n* niveles.

Ya que cada trabajo solo puede ser asignado a un funcionario, llevaremos cuenta de los trabajos ya asignados en

un vector marcador *asignado*[1..n] de booleanos.

Apartado (a)-----------------------------

En este apartado el problema consiste en elegir los x_i tales que se minimice $\sum_{i=1}^{n} T[i, x_i]$.

Veamos cómo podemos obtener una estimación del coste total a partir del coste de una solución parcial que nos ayude a podar nodos. Para la solución parcial $(x_1,\ldots,x_k)$, el tiempo hasta el momento es $\text{tiempo} = \sum_{i=1}^{k} T[i, x_i]$, y tenemos que estimar el tiempo del resto de la solución. La opción más sencilla (y más optimista) es aproximar con 0 este tiempo, y utilizar *tiempo* como estimación. Sin embargo, esta solución es muy poco afinada, por lo que la poda será poco eficiente.

Otra posibilidad es calcular un mínimo global de la matriz *T*,

$mínT = \text{mín}\{T[i, j] \mid 1 \le i \le n \wedge 1 \le j \le n\}$.

que sirva como cota inferior al tiempo de realización de cada trabajo por los funcionarios (*como se hace en la solución del Ejercicio 14.14 al minimizar el coste de una arista del grafo*). Así podemos aproximar el tiempo del resto de la solución con $(n-k)\ mínT$. Este cálculo de la aproximación es eficiente ya que el mínimo global solo se calcula una vez (con coste en $\Theta(n^2)$) y solo se mantiene una variable (espacio adicional), pero la estimación sigue siendo poco afinada.

La siguiente alternativa, más razonable, consiste en tener calculado un

mínimo por cada fila: tendremos calculado, para cada funcionario, cuánto tarda en realizar el trabajo que realiza más rápidamente:

rápido[i]= mín{*T*[i, *j]* | 1≤*j*≤n).

De esta manera podemos calcular la estimación del tiempo restante como:

$$\sum_{i=k+1}^{n} \textit{rápido}\ [i].$$

El vector *rápido* se inicializa al principio con un coste cuadrático, pero ahora el cálculo de las cotas es lineal (aunque se mejorará).

Al calcular el vector *rápido* al principio del algoritmo, no tenemos en cuenta que ya hay trabajos repartidos, cuando estamos en una solución parcial determinada, que no deberían entrar en el cálculo de la aproximación correspondiente. Podemos afinar la estimación, más todavía, calculando *rápido* dinámicamente entre los trabajos no repartidos ya a algún funcionario:

rápido-din[i] = mín{T[i,*j*] | 1≤*j*≤*n* ∧ *j* no ha sido asignado}

pero esto haría que el cálculo de la estimación pasara a ser cuadrático aun suponiendo que, mediante un marcador, sepamos en tiempo constante qué trabajos se han repartido y cuáles no.

Implementamos, por tanto, la opción intermedia en la que el vector *rápido* se calcula al principio del algoritmo. Además calcularemos inicialmente (y, por tanto, solo una vez) las sumas

$$\text{est}[k] = \sum_{i=k+1}^{n} \text{rápido } [i]$$

de tal forma que no tengan que ser recalculados dentro del algoritmo.

El algoritmo recursivo queda así:

```
proc funcionariosMínVa(e T[1..n] de real+, e
est[1 ..n] de real, sol[1..n] de 1..n, e k: 1..n,
tiempo: real, asignado[1..n] de bool,
solMejor[1..n] de 1..n, tiempoMejor: real∞)

para t=1 hasta n hacer
  si ¬asignado[t] entonces
    sol[k]:= t;
    asignado[t]:= cierto;
    tiempo:= tiempo + T[k, sol[k]];   {marcar}
    tiempoEstimado:= tiempo + est[k];
    si tiempoEstimado < tiempoMejor
    entonces   {se puede mejorar}
      si k=n entonces
        solMejor:= sol;
        tiempoMejor:= tiempo
      si no
        funcionariosMínVa(T, est, sol, k+1, tiempo,
        asignado, solMejor, tiempoMejor);
      fsi
    fsi
    asignado[t]:= falso;
    tiempo:= tiempo - T[k, sol[k]] {desmarcar}
  fsi
fpara
fproc
```

La función que realiza el precálculo de las estimaciones es:

```
fun cálculoEstimacionesMin(T[1..n, 1..n] de
real+) dev est[1..n] de real
var rapido[1..n] de real+
```

```
{cálculo de los mínimos por cada fila de T}
para i = 1 hasta n hacer
   rápido[i] := T[i,1];
   para j = 2 hasta n hacer
      rápido[i] := mín(rapido[i], T[í, j]);
   fpara
fpara

{cálculo de las estimaciones}
est[n] : = 0;
para i = n - 1 hasta 1 paso - 1 hacer
   est[i]:= est[i+1] + rápido[i+1];
fpara
ffun
```

$$\left\{ \text{est}[k] = \sum_{i=k+1}^{n} \min_{1 \leq j \leq n} \{T[i,j]\} \right\}$$

La función principal que realiza la inicialización adecuada y busca la mejor solución es la siguiente:

```
fun funcionariosMín(T[1..n, 1..n] de real+)
dev ⟨solMejor[1..n] de 1..n, tiempoMejor:
real∞ ⟩
var est[1..n] de real, sol[1..n] de 1 ..n,
asignado[1..n] de bool

est[1..n]:= cálculoEstimacionesMín(T);
tiempo:= 0;
asignado[1..n] := [falso];
tiempoMejor +∞;
funcionariosMínVa(T, est, sol, 1, tiempo,
asignado, solMejor, tiempoMejor);
ffun
```

Apartado (b)-----------------------------

Este apartado es dual al anterior donde en vez de minimizar tenemos que *maximizar*: el problema consiste en elegir los xi tales que se maximice $\sum_{i=1}^{n} E[i, x_i]$.

Veamos cómo podemos utilizar las mismas técnicas para obtener una estimación a partir de la eficacia de una solución parcial que nos ayude a podar nodos. En este caso necesitamos una cota *superior* de la eficacia alcanzable a partir de una solución parcial. Cuando esta cota superior no supere la eficacia de la mejor solución encontrada, podemos podar la solución parcial actual.

Para la solución parcial (x_1,…,x_k), la eficacia hasta el momento es *eficacia* = $\sum_{i=1}^{n} E[i, x_i]$ y tenemos que estimar la eficacia del resto de la solución. Al contrario que en el apartado anterior, *no podemos* utilizar este valor como estimación, ya que es una cota inferior pero no superior.

En cambio, sí podemos calcular un máximo global de la matriz *E.*

máxE = máx{E[i, *j*] | 1≤i≤n ∧ 1 ≤*j*≤n),

y aproximar la eficacia del resto de la solución con (n-*k*) *máxE.*

También podemos tener calculado un máximo por cada fila calculando, para cada funcionario, la eficacia con la que consigue realizar el trabajo que mejor realiza.

mejor[i| = máx{E[i, j] | 1≤*j*≤*n*],

y entonces podemos calcular la

aproximación de la eficacia de la solución restante como

$$\sum_{i=k+1}^{n} \text{mejor } [i]$$

Y, por último, también podemos calcular el vector *mejor* dinámicamente entre los trabajos no repartidos ya a algún funcionario:

mejorDin[i] = máx{E[i, *j]* | 1≤*j*≤*n* ∧ *j* no ha sido asignado}.

Ya que las complejidades del cálculo de las cotas son las mismas que en el apartado anterior, implementamos la opción en la que el vector *mejor* se calcula al principio del algoritmo, calculando inicialmente las sumas

$$\text{est}[k] = \sum_{i=k+1}^{n} \text{mejor } [i]$$

El algoritmo, en este caso, queda como sigue (obsérvese la similitud con el apartado anterior):

proc funcionariosMáxVa(**e** E[1..n, 1..n] **de** *real*⁺**,**
e est[1..n] de *real, sol[* 1 ..71] de 1 *..11,* **e** *k: :* 1 *..n.*
eficacia : rea/, asig/iado[l..n] de *bool,*
sol-mejor[l..n] **de** 1..*n. eficacia-mejor : real^)*

para *t* = 1 **hasta** *n* **hacer**
si -'flSíg/mdo}/] **entonces**
ío/[A] := *t*
asignado[t] := cierto: *eficacia* := *eficacia* + *E[k. sol[k}]* {marcar}
eficacia-estimada := *eficacia* + <?.s7[X:]
si *eficacia-estimada* > *eficacia-mejor* **entonces**
{ se puede mejorar }
si *k* — *11* **entonces**
sol-mejor := *sol* ; *eficacia-mejor* := *eficacia*

```
si no
funcionarios-máx-va(E. est. sol. k + 1. eficacia,
asignado, sol-mejor, eficacia-mejor) fsi
fsi
asignado[t] := falso ; eficacia := eficacia — E[k,
sol[k]] j desmarcar }
fsi
  fpara
fproc
```

La función que realiza el precálculo de las estimaciones es:

```
fun cálculo-estimaciones-máx(E[l..n, I..71J
de rea/+) dev es/[l..n] de real
var /íie/or[l..n] de real+
{cálculo de los máximos por cada fila de E}
para i = 1 hasta n hacer
mejor[i] := E[i, 1]
para j = 2 hasta 7; hacer
mejor[i] := máx(me/'or[í], E[í, J])
fpara
fpara
{cálculo de las estimaciones}
es/[n] := 0
para i — n — 1 hasta 1 paso - 1 hacer
esr[í] := est[i + 1] + inejor[i + 1]
fpara
ffun
```

(«4*J = E;U+ı máx]<;<„{£ [ı, ;]))

Y la función principal que realiza la inicialización adecuada y busca la mejor solución es la siguiente:

```
fun funcionariosMáx(E[l..n, l..n] de
rea/+) dev {sol-mejor[ 1 ,.n] de 1 ..n.
eficacia-mejor : rea/^ ) var e.sr[l../i| de
rea!, so/[l..n] de I..11, asignado[ 1..71]
de bool
    es/[l.jí] := cálculo-estimaciones-máxff)
    eficacia := 0
```

asignadol I ..zi] := [falso]
eficacia-mejor := —00
funcionarios-máx-va(£, *est. sol.* 1. *eficacia, asignado, sol-mejor, eficacia-mejor'*)
ffun

-14.16. Huertas

El tío Facundo posee *n* huertas, cada una con un tipo diferente de árboles frutales. Las frutas ya han madurado y es hora de recolectarlas. El tío Facundo conoce, para cada una de las huertas, el beneficio *b,* que obtendría por la venta de lo recolectado. El tiempo que se tarda en recoger los frutos de cada finca es asimismo variable y viene dado por ti. También sabe los días di que tardan en pudrirse los frutos de cada huerta. Ayudar al tío Facundo a decidir qué debe recolectar para maximizar el beneficio total obtenido.

------------------------Solución--------------------------

Este ejercicio se resolvió, para el caso particular en el que la recolección de cada huerta dura un día, mediante el método voraz en el Ejercicio 12.11.

Tenemos que seleccionar un subconjunto de las huertas de tal forma que las seleccionadas puedan terminar de recolectarse antes de su plazo y el beneficio obtenido sea máximo. Podemos representar las soluciones mediante tuplas $(x_1,\ldots,x_n)$ donde $x_i=1$ indica que los frutos de la huerta *i* se recolectarán mientras que $x_i=0$ indica que los frutos de la huerta *i* no se recolectarán, es decir, un subconjunto de huertas se representa mediante su función característica.

El árbol de exploración, binario, es análogo al de la Figura 14.8.

Para que un conjunto de huertas sea

factible, la recolección de sus frutos tiene que poder organizarse de forma que en cada una de las huertas se termine sin superar el correspondiente límite de días (fecha de caducidad). Más formalmente, si tomamos las huertas $i_1,\ldots,i_m$ en dicho orden, el tiempo que se tarda en terminar de recolectar los frutos de la huerta ik es $\sum_{j=1}^{k} t_{ij}$ de modo que la secuencia es *admisible* si $\sum_{i=1}^{k} t_{ij} \leq d_{i_k}$ para todo k entre 1 y m. Para comprobar si un conjunto es factible tomamos como referencia el problema que resolvimos mediante el método voraz (Ejercicio 12.11). Podemos utilizar una propiedad parecida a la que vimos para dicho problema:

Lema:

Dado un conjunto *H* de huertas, H es ***factible*** ⇔ la secuencia de huertas *H,* ordenadas de forma creciente según fecha de caducidad, es ***admisible***.

Demostración:

La implicación hacia la izquierda es trivial, así que vamos a demostrar la implicación hacia la derecha, demostrando el contrarrecíproco, es decir, que si la secuencia ordenada no es admisible es porque *H* no puede ser factible. Supongamos $H = \{h_1,\ldots,h_k\}$ con $d_1 \leq <\ldots \leq d_k$, y supongamos que la secuencia $h_1,\ldots,h_k$ no es admisible, es decir, existe alguna huerta h_r con $1 \leq r \leq k$ tal que $d_r < \sum_{i=1}^{r} t_i = T_r$. Pero entonces $\forall i$: $1 \leq i \leq r : di < T_r$ por lo que es imposible recolectar a tiempo todas las huertas $h_1,\ldots,h_r$. cualquiera que sea el orden en el que lo intentemos y por tanto *H* no es factible.

Esta propiedad nos garantiza que si partimos de un conjunto de huertas ordenado de forma creciente según fechas de caducidad, entonces basta comprobar que la última huerta se ha recolectado dentro de su plazo, suponiendo que las huertas se han ido recolectando en orden.

Para hacer el test en la etapa *k* necesitamos calcular

$$tiempo = \Sigma\{t_i \mid 1 \le i \le k \wedge \text{huerta } i \text{ elegida } (x_i=1)\}$$

y comprobar que $tiempo \le d_k$; para la decisión $x_k=0$ de no recolectar no hace falta ningún test.

Podríamos utilizar como aproximación del beneficio alcanzable a partir de la solución parcial actual la suma de los beneficios de las huertas que no hemos considerado, pensando que vamos a ser capaces de recolectarlas todas. Sin embargo, como sabemos que las huertas *i* tales que $tiempo + t_i > d_i$ no se podrán recolectar, tomamos como estimación del beneficio restante la suma de los beneficios de las huertas que todavía podrían recolectarse individualmente a partir del tiempo acumulado *tiempo* sin superar su plazo.

Con todas estas consideraciones, el algoritmo queda como sigue:

{D[1]≤ … ≤D[n]}

```
proc huertasVa(e T[1..n], D[1..n], B[1..n]
de real+, sol[1..n] de 0..1, e k: 1..n,
tiempo, beneficio: real, solMejor[1..n] de
```

```
0..1, beneficioMejor: real∞)

{hijo izquierdo — recolectar}
sol[k]:= 1;
si tiempo + T[k] ≤ D[k] entonces {es factible}
  tiempo:= tiempo + T[k];
  beneficio:= beneficio + B[k]; {marcar}
  beneficioEst:= cálculoEstimación(T, D, B, k,
  tiempo, beneficio);
  si beneficioEst > beneficioMejor entonces
    si k=n entonces
      solMejor:= sol;
      beneficioMejor:= beneficio;
    si no
      huertasVa(T, D, B, sol, k+1, tiempo, beneficio,
      solMejor, beneficioMejor)
    fsi
  fsi
  tiempo:= tiempo - T[k];
  beneficio:= beneficio - B[k] {desmarcar}
fsi

{hijo derecho — no recolectar}
sol[k]:= 0
{no hace falta test de factibilidad}
beneficioEst:= cálculoEstimación(T, D, B, k,
tiempo, beneficio)
si beneficioEst > beneficioMejor entonces
{se puede mejorar}
  si k=n entonces
  solMejor:= sol;
  beneficioMejor:= beneficio
  si no
  huertasVa(T, D, B, sol, k+1, tiempo, beneficio,
  solMejor, beneficioMejor)
  fsi
fsi
fproc
```

La función que calcula la estimación es:

```
fun cálculoEstimación(T[1..n], D[1..n],
```

```
B[1..n] de real+, k: 1 ..n, tiempo, beneficio:
real) dev beneficioEst: real

beneficioEst:= beneficio;
para i = k+1 hasta n hacer
   si tiempo + T[i] ≤ D[i] entonces
      beneficioEst:= beneficioEst + B[i]
   fsi
fpara
ffun
```

Y la función principal que devuelve la mejor solución es la siguiente:

```
fun huertasPrincipal(T[1..n], D[1..n],
B[1..n] de real+) dev ⟨solMejor[1..n] de
0..1, beneficioMejor: real⟩
var sol[1..n] de 0..1

tiempo:= 0;
beneficio:= 0;
beneficioMejor:= -1; {peor que cualquier
solución}
huertasVa(T, D, B, sol, 1, tiempo, beneficio,
solMejor, beneficioMejor)
ffun
```

-14.17. Componentes electrónicos

Tenemos un conjunto de *n* componentes electrónicos para colocar en *n* posiciones sobre una placa. Se nos dan 2 matrices de dimensiones *n* x *n. N* y *D.* donde *N[i,* j] indica el número de conexiones necesarias entre la componente *i* y la componente *j,* y *D[p, q*] indica la distancia sobre la placa entre la posición *p y* la posición *q* (ambas matrices son simétricas y con diagonales nulas). Un cableado $(x_1,\dots,x_n)$ de la placa consiste en la colocación de cada componente *i* en una posición distinta x,. La longitud total de este cableado viene dada por la fórmula:

$$\sum_{i<j} N[i,j]D\left[x_i, x_j\right]$$

Escribir un algoritmo para encontrar el cableado de longitud mínima.

------------------------Solución------------------------

Este ejercicio es muy parecido al Apartado (a) del Ejercicio 14.15, aunque con una función de coste más complicada.

Las soluciones al problema son los posibles cableados $(x_1,\dots,x_n)$, donde x_i es la posición asignada a la componente *i.* Se tiene que cumplir que las posiciones asignadas sean válidas, y que no se repitan, es decir, son permutaciones de {1,...,n}. El árbol de exploración es similar al del Ejercicio 14.1, con *n* niveles. Mantendremos marcadores con las posiciones ya utilizadas y el coste del

cableado que representa la solución parcial en la que nos encontramos.

Veamos cómo calcular una estimación del coste mejor alcanzable a partir de la solución parcial actual. La función de coste es

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} N[i,j]D[x_i,x_j] = \sum_{j=2}^{n} \sum_{i=1}^{j-1} N[i,j]D[x_i,x_j]$$

Antes de tomar la decisión en la etapa *k* se tiene el coste acumulado

$$\text{coste} = \sum_{j=2}^{k-1} \sum_{i=1}^{j-1} N[i,j]D[x_i,x_j]$$

y al tomar la decisión para xk, pasamos a tener

$$\sum_{j=2}^{k} \sum_{i=1}^{j-1} N[i,j]D[x_i,x_j]$$

$$= \text{coste} + \sum_{i=1}^{k-1} N[i,k]D[x_i,x_k].$$

Por tanto, la actualización del coste parcial acumulado requiere un bucle lineal con respecto a *k.*

Para estimar el resto, precalculamos el mínimo global en la matriz *D, mínD.* Entonces

coste-estimado

$$= \text{coste} + \sum_{j=k+1}^{n} \sum_{i=1}^{j-1} N[i,j]minD$$

$$= \text{coste} + minD \sum_{j=k+1}^{n} \sum_{i=1}^{j-1} N[i,j]$$

donde la suma que aparece también se

puede precalcular ya que solamente depende del valor de *k* y de los datos dados en la matriz *N.*

El procedimiento recursivo es el siguiente:

```
proc cableado-va(e N[1..n, 1..n] de nat, e
D[1..n, 1..n] de real, e est[1..n] de real,
sol[1..n] de 1 ..n, e k: 1 ..n, coste : real,
usada[1..n] de bool, sol-mejor[l..n] de 1..n,
coste-mejor : real∞)
para p = 1 hasta n hacer
si ¬ usada[p] entonces
sol[k] := p
{marcar}
usada[p] := cierto
coste-comp := coste-componente(N,D,sol,k)
coste := coste + coste-comp
coste-estimado := coste + est[k]
si coste-estimado < coste-mejor entonces {se
puede mejorar sol-mejor}
si k = n entonces
sol-mejor := sol ; coste-mejor := coste
si no cableado-va( A', D. est, sol. k + 1, coste,
usada, sol-mejor, coste-mejor)
fsi
fsi
{desmarcar}
coste := coste - coste-comp
usada[p] := falso
fsi
fpara
fproc
```

Las funciones que calculan el coste de añadir una nueva componente y las estimaciones iniciales son:

```
fun costeComponente (N[1..n, 1..n] de nat,
D[1..n, 1..n] de real, sol[1..n] de 1..n, k: 1..n)
dev costeComp : real

costeComp := 0
  para i = 1 hasta k -1 hacer
  costeComp := costeComp + N[i, k] * D[sol[i],
```

```
sol[k]]
fpara
ffun
```

```
fun preCálculoEstimaciones(N[1..n, 1..n] de
nat, D[1..n, 1..n] de real) dev est[1..n] de real

//{cálculo del mínimo de D}
mínD := +∞
para i = 1 hasta n hacer
  para j = i + 1 hasta n hacer
  mínD = mín(mínD, D[i, j])
  fpara
fpara

//{cálculo de las estimaciones}
est[n] := 0
para k = n - 1 hasta 1 paso - 1 hacer
  est[k] := est[k+1]
   para i = 1 hasta k hacer
   est[k] := est[k] + N[i, k+1]
     fpara
fpara

para k = 1 hasta n hacer
est[k] := mínD * est[k]
fpara
ffun
```

$$\left\{ \text{est}[k] = minD \sum_{j=k+1}^{n} \sum_{i=1}^{j-1} N[i,j] \right\}$$

Y la función principal que busca la mejor solución es:

```
fun cableado(N[1..n, 1..n] de nat, D[1..n, 1..n]
de real) dev (solMejor[1..n] de 1..n, costeMejor:
real∞ )
var est[1..n] de real, sol[1..n] de 1..n,
usada[1..n] de bool

est[1..n] := preCálculoEstimaciones(N, D);
coste := 0;
```

```
usada[1..n] := [falso];
costeMejor := +∞
cableadoVal(N, D, est, sol, 1, coste, usada,
solMejor, costeMejor}
ffun
```

-14.18. Parejas “afines”

Un grupo de amigos, formado por *n* parejas (*n* mujeres y *n* hombres), se reúnen para cenar. A la hora de sentarse a la mesa (redonda), hombres y mujeres deben alternarse, y nadie debe sentarse ai lado de su pareja habitual. Hay que maximizar el grado de bienestar total, obtenido sumando los grados de afinidad mutuos entre los comensales sentados en posiciones adyacentes. Al efecto, se dispone de sendas matrices que nos indican la afinidad entre hombres y mujeres, y entre mujeres y hombres. Una vez establecida cada pareja de vecinos, la afinidad mutua se calcula multiplicando la de cada miembro por la del contrario. Diseñar un algoritmo que encuentre una solución óptima al problema.

---------------------------Solución-------------------------

Supongamos las sillas numeradas de 1 a *2n.* Ya que se tienen que alternar hombres y mujeres, fijamos que en las posiciones impares se sentarán hombres y en las pares mujeres. Podemos representar las soluciones como tuplas $(x_1,\ldots,x_{2n})$ donde $x_i = j$ indica que en la silla *i* se sienta el hombre de la pareja *j* si *i* es impar, o la mujer de la pareja *j* si *i* es par. Se tiene que cumplir que nadie se siente al lado de su pareja, es decir, en la solución no puede haber números consecutivos iguales. Dada la estructura circular de las soluciones, para evitar generar soluciones repetidas, fijamos que en la primera silla se sienta el hombre de

la primera pareja, x_1=1 (véanse las soluciones de los Ejercicios 14.9 y 14.14). El árbol de exploración, con *2n* niveles, es de la forma mostrada en la Figura 14.10.

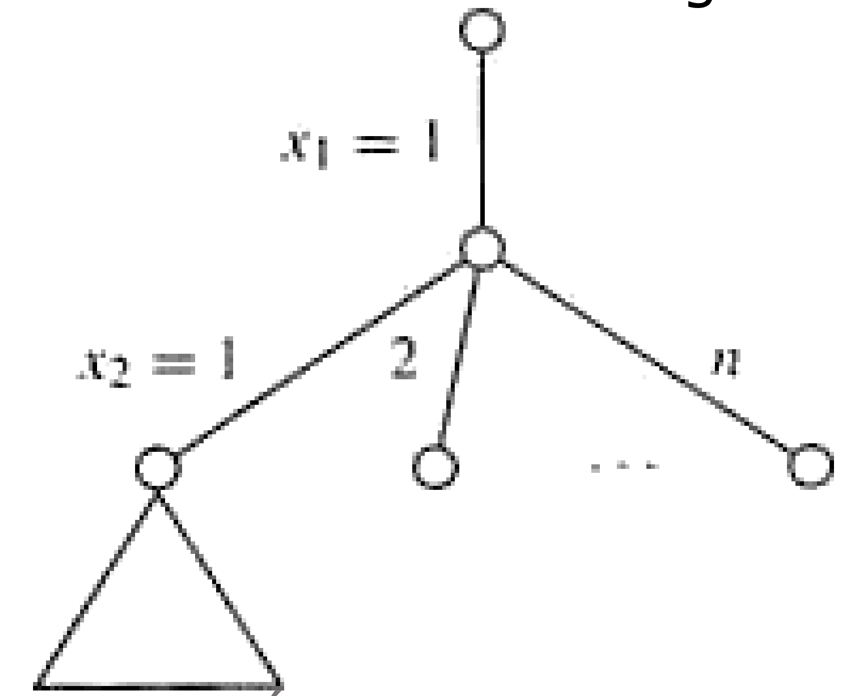


Figura 14.10: Árbol de exploración para el problema de los comensales.

Utilizaremos *sexo*=0 para referirnos a las mujeres y *sexo*=1 para los hombres. Para hacer más eficiente la comprobación de que no sentamos varias veces a la mesa a la misma persona, tendremos como marcador una matriz *sentado[0..* 1, 1..*n]* de booleanos, que indicará si la correspondiente persona ha sido ya incluida en la solución parcial o no:

$$\forall i: 1 \leq i \leq n: \begin{cases} \text{sentado } [0, i] \Leftrightarrow \text{ la mujer de la pareja } i \text{ aparece en sol } [1..k] \\ \text{sentado } [1, i] \Leftrightarrow \text{ el hombre de la pareja } i \text{ aparece en sol } [1 \ldots k]. \end{cases}$$

Supondremos que las preferencias vienen dadas en una tabla P[0..1, 1..n, 1..n] donde P[0, i, *j]* indica la preferencia de la mujer de la pareja *i* por el hombre de la pareja *j,* y P[1, *i,* j] la preferencia del hombre de la pareja *i* por la mujer de la pareja *j.* De esa manera, la función de bienestar total cuando todos están

sentados a la mesa es

$$\left(\sum_{i=2}^{2n} P[s, x_i, x_{i-1}]P[\bar{s}, x_{i-1}, x_i]\right) + P[1, x_1, x_{2n}]P[0, x_{2n}, x_1]$$

donde
s = *i* mod 2
$\bar{s} = 1 - s$.

Cuando hemos tomado decisiones hasta x_k *(k < 2n)* tenemos un bienestar acumulado:

$$\text{bienestar} = \sum_{i=2}^{k} P[s, x_i, x_{i-1}]P[\bar{s}, x_{i-1}, x_i]$$

Para estimar el resto, precalculamos los máximos globales de las matrices P[0, *i, j*] y P[1, *i,* j], que denotaremos por *máxM* y *máxH*, respectivamente (y que se pueden calcular antes de hacer la llamada inicial). Entonces

$$\begin{aligned}\text{bienestarEstimado} &= \text{bienestar} + \left(\sum_{i=k+1}^{2n} \textit{máxH}\ \text{máxM}\right) + \text{máxH máxM}\\ &= \text{bienestar} + (2n - k + 1)\ \text{máxH máxM}.\end{aligned}$$

El algoritmo recursivo de vuelta atrás es:

proc **cenaVA**(**e** P[0..1, 1..*n*, 1..n] **de** *real*, **e** *máxH*, *máxM* : *real*, *sol*[1..2n] **de** 1..*n*, **e** *k:* 1..2n, *bienestar: real*, *sentado[0..1*, 1..n] **de** *bool*, *solMejor[1..2n]* **de** 1..n, *bienestarMejor*: real$_\infty$)

sexo := *k* mod 2;
anterior := *sol*[*k* - 1];
para *i* = 1 **hasta** *n* **hacer**
si ¬*sentado[sexo, i]* ∧ *i* ≠ *anterior* **entonces**
sol[k] := *i*

```
{marcar}
sentado[sexo, i] := cierto;
bienestar := bienestar + P[sexo, i, anterior] *
P[1 - sexo, anterior, i];
si k = 2 * n entonces
  si i ≠ 1 ∧ bienestar + P[1, 1, i] * P[0, i, 1] >
  bienestarMejor entonces
    solMejor := sol;
    bienestarMejor:= bienestar + P[1, 1, i]* P[0,
    i, 1]
  fsi
si no
  bienestarEstimado := bienestar + (2 * n - k +
  1) * máxH * máxM;
  si bienestarEstimado > bienestarMejor
  entonces   {se puede mejorar}
  cenaVA(P, máxH, maxM, sol, k+1, bienestar,
  sentado, solMejor, bienestarMejor);
  fsi
fsi
{desmarcar}
sentado[sexo, i] := falso;
bienestar := bienestar - P[sexo, i, anterior] *
P[1 - sexo, anterior, i];
fsi
fpara
fproc
```

La función principal, que calcula los máximos globales, hace la inicialización y llama al procedimiento anterior para buscar la mejor solución, es la siguiente:

```
fun cena(P[0..1, 1..n. 1..n] de real) dev
⟨solMejor[1..2n] de 1..n, bienestarMejor: real∞⟩
var sol[1..2n] de 1..n, sentado]0.. 1, L.n] de
bool

{cálculo de los máximos}
máxH := -∞;
máxM := -∞;
para i = 1 hasta n hacer
```

```
    para j = 1 hasta n hacer
      si i ≠ j entonces
        máxH := máx(máxH, P[1,i,j]);
        máxM := máx{máxM, P[0, i, j]);
      fsi
    fpara
  fpara
  sol[1]:= 1;
  bienestar := 0;
  sentado[0..1, 1..n] := [falso];
  sentado[1,1] := cierto;
  bienestarMejor := -∞;
  cenaVA(P, máxH, máxM, sol, 2, bienestar,
  sentado, solMejor, bienestarMejor)
ffun
```

Nótese que como se empieza con k=2, el acceso a *sol*[k-1] siempre va a ser correcto.

A la hora de calcular los máximos no hemos tenido en cuenta las preferencias de un hombre o una mujer por su propia pareja, ya que nunca se van a sentar juntas dos personas de la misma pareja. Una alternativa hubiera sido exigir que $\forall i: 1 \leq i \leq n$: P[0,i,i]= P[1,$i$,$i$] = -∞.

-14.19. Mochila Ali Baba

Cuando Alí-Babá consigue entrar en la Cueva de los 40 Ladrones encuentra allí objetos valiosos. Alí-Babá conoce el peso y valor (números reales) de cada uno de los objetos. Solo puede llevar consigo los objetos que quepan en su pequeña mochila, que soporta un peso máximo M conocido. Determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

---------------------------Solución-------------------------

(Este mismo problema fue resuelto mediante el método de *programación dinámica* en el Ejercicio **13.2**, y una variante donde los objetos se pueden *fraccionar* mediante el método *voraz* en el Ejercicio 12.5).

Supongamos que en la cueva hay *n* objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$, y consideremos que la mochila soporta un peso total máximo *M* > 0.

El problema consiste en maximizar $\sum_{i=1}^{n} x_i v_i$ con la restricción $\sum_{i=1}^{n} x_i p_i \leq M$ donde $x_i \in \{0, 1\}$ indica si hemos cogido (1) o no (0) el objeto *i.* De modo que las posibles soluciones son las tuplas $(x_1,\ldots,x_n)$, y el árbol de exploración, binario, es similar al de la Figura 14.8.

Vamos a llevar marcadores *peso* y *beneficio* con el peso y el beneficio de la solución parcial actual, que se van calculando incrementalmente.

Vamos a desestimar una solución parcial por cualquiera de las siguientes 2 razones: porque no sea factible (es decir, se haya excedido el peso máximo *M* de la mochila), o porque no interese extenderla (porque sepamos que cualquier extensión va a ser peor que la solución mejor encontrada hasta el momento).

Como el problema es de maximización, tenemos que encontrar una cota superior de la mejor solución alcanzable. Esta cota superior la vamos a obtener usando el algoritmo voraz (Ejercicio 12.5) que resolvía este mismo problema cuando los objetos se podían fraccionar ($0 \leq x_i \leq 1$). Ya que esta solución era óptima, no puede haber ninguna solución sin fraccionar objetos que sea mejor que la obtenible cuando se nos permitía hacerlo. Para utilizar este algoritmo necesitamos que los objetos estén ordenados en orden decreciente de valor por unidad de peso, *vi /pi*. Si estamos en una solución parcial donde se han tomado decisiones sobre los objetos del 1 al *k*, entonces la cota superior se calculará usando el algoritmo voraz para decidir cuáles hay que tomar, y qué beneficio se obtiene, de los objetos del k+1 al n, permitiendo $0 \leq x_j \leq 1$ cuando $k+1 \leq j \leq n$.

Ya que pasar al hijo izquierdo de un nodo consiste en tomar un objeto más (x_k = 1), la cota para un nodo y para su hijo izquierdo (si este es factible) es la

misma. Por tanto, solo hace falta recalcular la cota cuando pasemos a un hijo derecho y dejemos de coger un objeto. Además, como al pasar al hijo izquierdo tampoco cambia la mejor solución calculada hasta el momento, no hace falta repetir el test que pregunta si interesa expandir; en particular, si se llega a $k=n$ es seguro que hemos encontrado una solución mejor, que se puede actualizar.

El algoritmo es el siguiente:

$\{\frac{P[1]}{D[1]} \geq \cdots \geq \frac{P[n]}{D[n]}\}$

```
proc mochilaVa(e P(1.,n], V[1..n] de real⁺, e M: real, sol[1..n] de 0..1, e k: 1..n, peso,
beneficio: real, solMejor[1..n] de 0..1,
beneficioMejor: real)

{hijo izquierdo — coger objeto, no hacemos estimación}
sol[k] := 1;
peso := peso + P[k];
beneficio := beneficio + V[k];  {marcar}
si peso ≤ M entonces
   si k=n entonces
      solMejor := sol;
      beneficioMejor := beneficio;
   si no
      mochilaVa(P, V, M, sol, k+1, peso,
      beneficio, solMejor, beneficioMejor);
   fsi
fsi
peso := peso - P[k] ;
beneficio := beneficio – V[k] {desmarcar}
{hijo derecho — no coger objeto, no se marca pero sí se hace estimación}
sol[k] := 0;
beneficioEstimado := cálculoEstimación(P, V, M,
k, peso, beneficio)
si beneficioEstimado > beneficioMejor entonces
   si k=n entonces
```

```
      solMejor := sol;
      beneficioMejor := beneficio
    si no
      mochilaVa(P, V, M, sol, k+1, peso,
      beneficio, solMejor, beneficioMejor);
    fsi
  fsi
fproc
```

$$\left\{\frac{V[1]}{P[1]} \geq \cdots \geq \frac{V[n]}{P[n]}\right\}$$

```
fun cálculoEstimación(P[1..n], V[1..n] de real+,
M: real+, k: 1..n, peso, beneficio: real) dev
estimación: real

hueco := M - peso;
estimación := beneficio;
j := k + 1;
mientras j≤n ∧ P[j]≤hueco hacer
{podemos coger el objeto j entero}
hueco := hueco – P[j];
estimación := estimación + V[j];
j := j + 1;
fmientras
si j ≤ n entonces {quedan objetos por probar}
{fraccionamos el objeto J (solución voraz)}
estimación := estimación + (hueco/ P[j]) * V[j];
fsi
ffun
```

La función principal es:

```
fun mochilaPrincipal(P[1..n], V[1..n] de real+,
M: real+) dev ⟨solMejor[1..n] de 0.. 1,
beneficioMejor: real⟩
var sol[1..n] de 0.. 1

peso := 0 ;
beneficio := 0;
beneficioMejor := - 1 {peor que cualquier solución}
mochilaVa(P, V, M, sol, 1, peso, beneficio,
solMejor, beneficioMejor);
ffun
```

Tengo un libro que dice que este problema se puede resolver mediante vuelta atrás y mediante programación dinámica, además de mediante método voraz si los objetos fuesen fraccionables. ¿Es verdad?

Sí, es verdad. El problema que has descrito es una variante del clásico problema de la **MOCHILA**, y dependiendo de las características específicas del problema, puede resolverse mediante diferentes enfoques, incluidos vuelta atrás, programación dinámica y método voraz. Veamos cómo se aplicaría cada uno de estos métodos:

1. Método Voraz (para objetos fraccionables)

Cuando se permite FRACCIONAR los objetos, (es decir, tomar partes de un objeto en lugar del objeto completo), el problema se convierte en la versión "fraccionaria" o "continua" del problema de la mochila. En esta versión, un enfoque voraz es óptimo. El algoritmo voraz consistiría en calcular el valor por unidad de peso (valor/peso) de cada objeto, y luego ir seleccionando objetos para añadir a la mochila comenzando por el que tenga **mayor** valor por unidad de peso, añadiendo tanto como sea posible hasta alcanzar el límite de peso de la mochila. Este enfoque asegura la maximización del valor dentro de la capacidad de peso permitida.

2. Programación Dinámica

La programación dinámica es una técnica poderosa para resolver el problema de la mochila cuando los objetos no son fraccionables, es decir, cada objeto debe ser tomado en su totalidad o dejado. Este enfoque es útil para la versión "**0-1**" del problema de la mochila, *donde cada objeto puede ser elegido o no*, pero no parcialmente. La programación dinámica resuelve el problema construyendo una tabla que representa soluciones a subproblemas de menor tamaño, y utiliza estos resultados para encontrar la solución óptima al

problema completo. Este método es especialmente efectivo ya que evita recalculaciones innecesarias y garantiza encontrar la solución óptima.

3. Vuelta Atrás (Backtracking)

El método de vuelta atrás es un enfoque de fuerza bruta mejorado que explora el espacio de todas las posibles combinaciones de objetos que podrían incluirse en la mochila, pero con la inteligencia de "volver atrás" y descartar rápidamente aquellas combinaciones que ya se sabe que no cumplirán con el requisito de peso máximo o que no serán óptimas. Aunque este método puede ser más lento que la programación dinámica para grandes instancias del problema debido a que explora muchas combinaciones, es capaz de encontrar la solución óptima y puede ser más intuitivo en algunos casos.

Conclusión

En resumen, tu libro está en lo correcto al afirmar que el problema puede ser abordado mediante vuelta atrás y programación dinámica para objetos no fraccionables, y mediante un método voraz si los objetos pueden ser fraccionados. La elección del método depende de la naturaleza específica del problema (si los objetos son fraccionables o no) y de consideraciones prácticas como la complejidad del algoritmo y el tamaño del problema.

-14.20. Cinta canciones

Pepe Casanova es un ligón de los de antaño, que intenta encandilar a las chicas con canciones románticas. A tal efecto, y de cara al veraneo en una playa del sur, decide conseguir una cinta para el radiocasete de su coche con las mejores canciones de amor. Pepe es muy peculiar en sus gustos, y además anda algo escaso de dinero, por lo que en lugar de comprar una de tantas recopilaciones que circulan por el mercado discográfico, quiere grabársela él mismo. Rebuscando entre sus viejos vinilos ha confeccionado la lista de sus *n* canciones favoritas, junto con la duración individual de cada una. Lamentablemente, su cinta (de 2 caras) de *T* minutos no tiene capacidad suficiente para contener todas las canciones, así que Pepe ha otorgado una puntuación a cada canción (cuanto más le gusta, mayor es la puntuación).

Ayuda a Pepe a conseguir la *mejor* cinta, teniendo en cuenta que las canciones escogidas han de caber enteras y no es admisible que una canción se corte a la mitad al final de una cara de la cinta.

----------------------------Solución----------------------------

Puesto que el orden de grabación de las canciones, dentro de cada cara, no influye en la puntuación, este problema es similar al de la *MOCHILA (Ejercicio 14.19*); la diferencia es que ahora tenemos 2 "mochilas": las 2 caras de la cinta. Así. las soluciones serán tuplas

(x_1,…,x_n) donde xi = 1 indica que la canción *i* se graba en la primera cara, xi = 2 indica que la canción *i* se graba en la segunda cara y xi = 0 indica que la canción *i* no se graba. El árbol de exploración, con *n* niveles, es de la forma mostrada en la Figura 14.11.

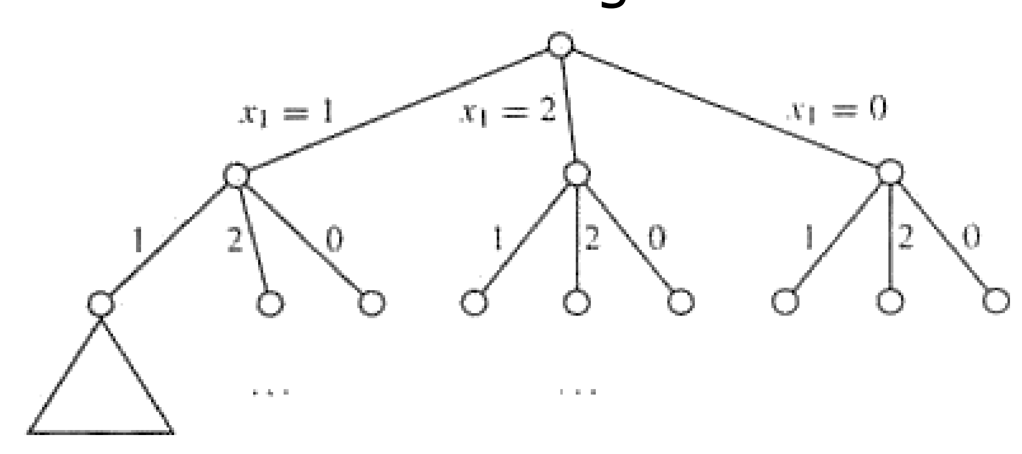


Figura 14.11: Árbol de exploración para el problema de las canciones.

La puntuación de una canción tampoco depende de la cara en la que se graba. Por tanto, para evitar generar soluciones equivalentes (con la misma puntuación total), solo consideraremos la posibilidad de grabar en la segunda cara cuando la ocupación actual de las 2 caras sea distinta.

Necesitaremos marcadores con el *beneficio* parcial y la ocupación en cada cara *ocupada*[1..2]. Podemos utilizar la misma estimación que en el caso de la mochila, considerando como espacio libre la suma del espacio libre en cada cara. Para ello necesitamos que las canciones estén ordenadas en order decreciente de puntuación por unidad de duración, *pi/di*.

Cuando probamos el primer hijo (x_k=1) de un nodo, si resulta factible, el beneficio estimado no cambia y,

además, la mejor solución encontrada hasta el momento es la misma, por lo que no es necesario repetir el test que decide si interesa expandir. Al pasar al segundo hijo (x_k=2). el beneficio estimadc no cambia, pero la mejor solución sí puede haber cambiado (al haber recorrido ya el primer subárbol) Aunque el beneficio mejor haya cambiado, nunca se puede haber hecho mayor que el beneficio estimado (cota superior), pero sí igual. Por tanto, al ser las comparaciones entre el beneficio estimado y el beneficie mejor estrictas, sí interesa volver a aplicar el test. Al pasar al tercer hijo (x_k=0) tanto el beneficie estimado como la mejor solución pueden cambiar, por lo que hay que repetir el test.

El algoritmo es el siguiente:

$\{\frac{P[1]}{D[1]} \geq \cdots \geq \frac{P[n]}{D[n]}\}$

proc cancionesVa(**e** D[1..n], P[1..n] de *real*$^+$, **e** *T*: *real*$^+$, *sol*[1..n] de 0..2, **e** *k*: 1..*n*, *ocupada[1..2]* de *real*, *beneficio*: *real*, *solMejor*[1..n] de 0..2, *beneficioMejor*: *real*)

{primer hijo — grabar la canción *k* en la cara 1}
sol[k] := 1;
ocupada[1] := *ocupada[1]* + D[k] ;
beneficio := *beneficio* + *P[k]* {marcar}
si *ocupada[1]* ≤ T/2 **entonces** {la estimación coincide con la del nodo padre}

```
    si k = n entonces
        solMejor := sol;
        beneficioMejor := beneficio;
    si no
        cancionesVA(D, P, T, sol, k+1, ocupada,
        beneficio, solMejor, beneficioMejor);
    fsi
fsi
ocupada[1] := ocupada[1] - D[k];
beneficio := beneficio – P[k];   {desmarcar}
si ocupada[1] ≠ ocupada[2] entonces
    { segundo hijo — grabar la canción k en la cara 2}
    sol[k] := 2;
    ocupada[2] := ocupada[2] + D[k];
    beneficio := beneficio + P[k];   {marcar}
    si ocupada[2] ≤ T/2 entonces
        beneficioEstimado := cálculoEstimación(D, P, T,
        k, beneficio, ocupada);
        si beneficioEstimado > beneficioMejor
        entonces
            si k = n entonces
                solMejor := sol;
                beneficioMejor := beneficio;
            si no
                cancionesVA(D, P, T, sol, k+1, ocupada,
                beneficio, solMejor, beneficioMejor);
            fsi
        fsi
    fsi
    ocupada[2]:= ocupada[2] - D[k];
    beneficio := beneficio – P[k];  {desmarcar}
fsi

{tercer hijo — no grabar la canción k (siempre es
factible)}
sol[k]:= 0;
beneficioEstimado := cálculoEstimación(D, P, T,
k, beneficio, ocupada);
si beneficioEstimado > beneficioMejor entonces
    si k=n entonces
        solMejor := sol;
        beneficioMejor:= beneficio;
    si no
        cancionesVA(D, P, T, sol, k+1, ocupada,
```

```
    beneficio, solMejor, beneficioMejor);
  fsi
fsi
fproc
```

$$\left\{\frac{P[1]}{D[1]} \geq \cdots \geq \frac{P[n]}{D[n]}\right\}$$

```
fun cálculoEstimación(D[1..n], P[1..n] de real⁺,
T: real⁺, k: 1..n, beneficio: real, ocupada[1..2]
de real) dev estimación: real

hueco := T - (ocupada[1] + ocupada[2])
estimación := beneficio;
j := k+1;
mientras j ≤ n ∧ D[j] < hueco hacer
  {podemos grabar la canción j entera}
  hueco : = hueco - D [j];
  estimación := estimación + P[j];
  j := j + 1;
fmientras

si j ≤ n entonces {quedan canciones por probar}
  {cortamos la canción j}
  estimación := estimación + (hueco/ D[j]) *
  P[j];
fsi
ffun
```

La función principal es:

$$\left\{\frac{P[1]}{D[1]} \geq \cdots \geq \frac{P[n]}{D[n]}\right\}$$

```
fun canciones(D[1..n], P[1..n] de real⁺, T:
real⁺) dev ⟨solMejor[1..n] de 0..2,
beneficioMejor: real⟩
var sol[1..n] de 0..2, ocupada[1..2] de real

ocupada[1..2] := [0];
beneficio := 0;
beneficioMejor := - 1;  {peor que cualquier solución}
cancionesVA(D, P, T, sol, 1, ocupada, beneficio,
solMejor, beneficioMejor);
ffun
```

-14.21. Franquear 5 sellos

Juanito está pasando unas vacaciones en Tombuctú. Juanito disfruta mandando postales de los exóticos lugares que visita, para que sus amigos rabien de envidia cuando las reciban. A tal efecto se ha pasado por la oficina de correos más cercana y se ha hecho con un lote de sellos de *n* valores diferentes, disponiendo de **3** sellos de cada valor. En correos también le han informado de las diferentes tarifas de franqueo de tarjetas postales, y le han explicado que en la esquina superior derecha de cada postal aparece un bloque remarcado destinado a su franqueo. Dicho bloque está dividido en **5** casillas, destinadas cada una a un sello, de forma que un franqueo solo es admisible si se alcanza la tarifa correspondiente y se cubren exactamente las 5 casillas.

a) Escribir un algoritmo para generar tocias las formas admisibles de franquear una postal de tarifa *T* si entendemos que el orden en el que aparecen los sellos en las casillas es significativo.
b) Escribir ahora un algoritmo para generar todas las formas admisibles de franquear una postal de tarifa *T* si el orden de los sellos ***no es significativo***.
c) Escribir un algoritmo para obtener una solución con el mínimo coste posible.

-----------------Solución-----------------

Si tenemos *n* tipos de sellos diferentes (cuyos valores vienen dados en *V*[1..n]), entonces un franqueo admisible puede representarse como una tupla (x_1,..., x_5), donde x_i es el tipo de sello que se ha colocado en la casilla *i*.

Dada una tarifa *T,* los franqueos admisibles tienen que cumplir: que solo se utilizan sellos de los tipos válidos, que se franquea una cantidad mayor o igual que *T* y que no se han utilizado más de 3 sellos de cada tipo.

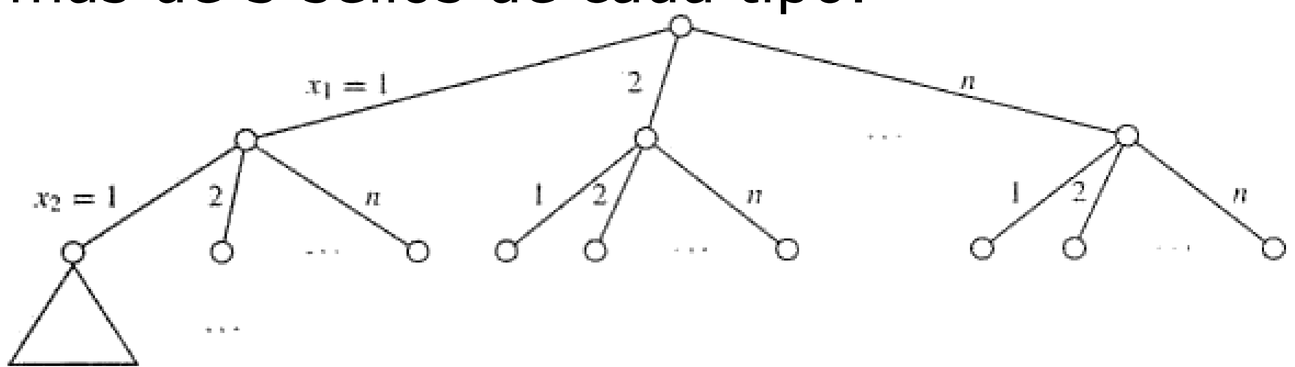


Figura 14.12: Árbol de exploración para el problema de los sellos (a).

Apartado (a)----------------

Como el orden de los sellos es importante, el árbol de exploración, con 5 niveles, y sin realizar ninguna poda, tiene la forma de la Figura 14.12.

Para comprobar que no utilizamos más de 3 sellos de cada tipo utilizamos un vector llamado *usados[1*..n] de números naturales, donde guardamos los sellos utilizados de cada tipo.

Además, para facilitar la comprobación de si un franqueo, supuestamente admisible, cubre la tarifa pedida, llevaremos la suma parcial de los sellos escogidos en otro marcador

llamado *coste*.

El algoritmo es el siguiente:

```
proc sellosVa1(e V[1..n] de real+, e T: real+, sol[1..5] de 1..n, e k: 1..5, coste :real, usados[1..n] de 0..3)

para s=1 hasta n hacer
  si usados[s] < 3 entonces
    sol[k] := s;
    usados[s] := usados[s] + 1;
    coste := coste + V[s] {marcar}
    si k=5 entonces
      si coste ≥ T entonces imprimir(sol) fsi
    si no
      sellosVa1(V, T, sol, k+1, coste, usados);
    fsi
    usados[s] := usados[s] - 1;
    coste := coste - V[s]; {desmarcar}
  fsi
fpara
fproc
```

Podríamos comprobar en cada nodo si hay suficientes sellos para colocar en las casillas vacías, y si los sellos no colocados tienen valores suficientemente grandes como para cubrir la tarifa restante. Sin embargo, en este apartado al ir construyendo soluciones nunca descartamos sellos, por lo que esta comprobación solo es necesaria al principio del algoritmo, en el procedimiento principal.

El procedimiento principal que realiza la inicialización adecuada e imprime todos los franqueos admisibles es:

```
proc sellos1(e V[1..n] de real⁺,e T: real⁺)
var sol[1..5] de 1..n, usados[1..n] de 0..3

sumaTodos := 0
para i = 1 hasta n hacer
    sumaTodos := sumaTodos + 3 * V[i];
fpara

si 3*n≥5 ∧ sumaTodos≥T entonces
    coste := 0;
    usados[1..n] := [0];
    sellosVa1(V, T, sol, 1, coste, usados);
fsi
fproc
```

La generalización correspondiente a disponer de *m* casillas y una cantidad c_i de sellos de tipo *i* es inmediata, tanto en este como en el resto de los apartados.

Apartado (b)-------------------------

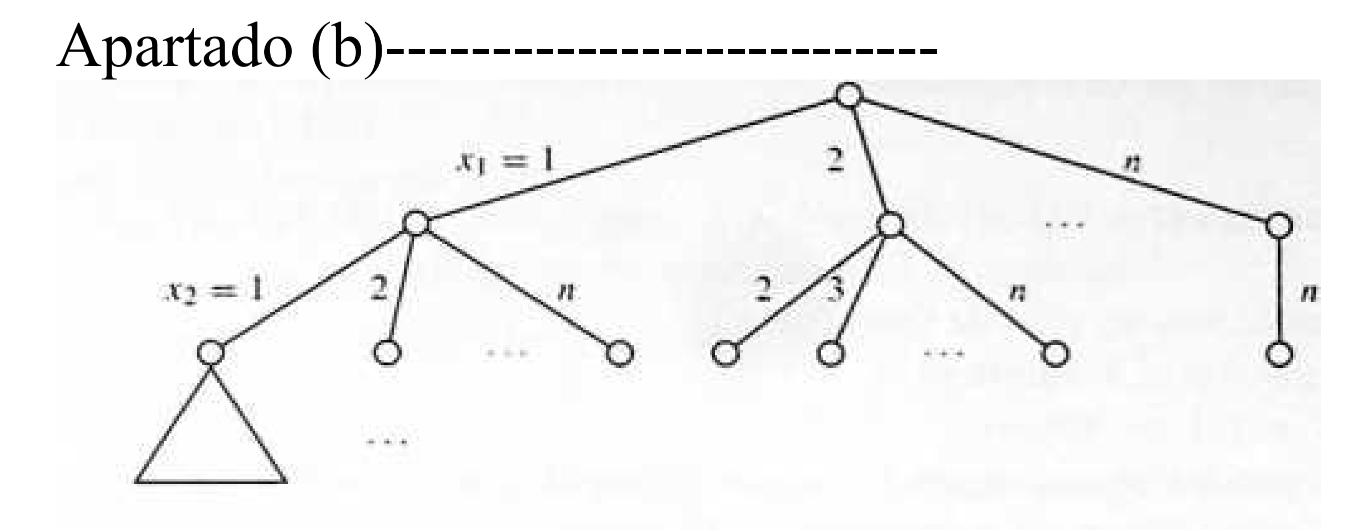


Figura 14.13: Árbol de exploración para el problema de los sellos (b

Como lo que importa en este caso son los valores escogidos y no la casilla que ocupan, impondremos un orden en la colocación de los sellos (por ejemplo, según la ordenación dada de los tipos de sellos). Así, antes de realizar ninguna poda, el árbol de exploración, con 5 niveles, tiene la forma mostrada en la Figura 14.13.

Ya que el número de hijos de cada nodo no es constante, llevaremos en la variable *último* el tipo del último sello colocado. Para comprobar que no hemos utilizado más de tres sellos de este último tipo, llevaremos una variable marcador *usados;* los restantes tipos de sellos o ya se han descartado o todavía no se han considerado. Al pasar a considerar un nuevo tipo de sellos, descartamos colocar los sellos no utilizados del tipo anterior. Por ejemplo, en el árbol de la Figura 14.13 el segundo hijo de la raíz indica que el primer sello utilizado es de tipo 2 y eso implica que en las soluciones en este subárbol no se considerará ningún sello de tipo 1.

Podemos, por tanto, podar más el árbol controlando en cada nodo que haya suficientes sellos para colocar en las casillas vacías y que se dispone de valores suficientemente grandes para cubrir la tarifa. Para esta última comprobación llevaremos en una variable *suma-libres* la suma de los valores de los sellos que no hemos usado ni hemos descartado usar.

El algoritmo es el siguiente:

```
proc sellosVa2(e V[1.JI] de real+. c T : real+.
sol] I ..5] de l.jt.ek . 1 ..5. último : 1..».
usados : 0..3. coste, suma-libres : real)
{ probamos un sello de tipo último }
si usados < 3 entonces
,vo/[A] := último
usados '.= usados + 1 ; coste := coste + V
[último] { marcar }
```

suma-libres suma-libres — V[último]
si *k* **= 5 entonces**
si *coste* > *T* **entonces** imprimirlo/) **fsi**
si no
si factible?(T, *k. último, usados, coste, suma-libres)* **entonces** sellos-va2(V, *T. sol. k* + 1. *último, usados, coste, suma-libres)* **fsi**
fsi
usados := *usados* — 1 : *coste* := *coste* — V[«W/no] { desmarcar) *suma-libres '.= suma-libres + V [último]* **fsi**
{descartamos los sellos no utilizados de tipo *último* |
suma-libres-act := *suma-libres* — (3 — *usados*) * *V[último]*
{probamos con el resto de tipos de sellos)
usados-act := 1
para s = *último* + 1 **hasta** *n* **hacer**
so/[/r] := s
coste := *coste* + V[s] (marcar)
suma-libres-act := *suma-libres-act* — V|s] (quitamos un sello de tipo ,v }
si *k* = 5 **entonces**
si *coste* > *T* **entonces** imprimir(soZ) **fsi**
si no
si factibleVíT. *k. s. usados-act, coste, suma-libres-act)* **entonces**
sellos-va2(V'. *T. sol, k* + 1,5, *usados-act, coste, suma-libres-act)*
fsi
fsi
suma-libres-act := *suma-libres-act* + *V[s]*
coste := *coste* — V[s] (desmarcar)
(quitamos los dos sellos restantes de tipo 5)
suma-libres-act := *suma-libres-act* — 2 * V[s]
fpara
fproc

Hemos utilizado una variable local *usados-act* en vez de *usados* porque el tipo de asignación que hacemos (asignar el valor 1) no permite un posterior desmarcado (recuperar el

valor antiguo de *usados).* También necesitamos una variable local *suma-libres-act* para facilitar al pasar de un hijo al siguiente hermano que se vaya acumulando el valor de los sellos descartados y recuperar el valor de *suma-libres* al volver al padre.

El valor de la variable *último* coincide con *sol[k* — 1], excepto en la primera llamada, donde esa posición del vector no existe. Podríamos eliminar esta variable extendiendo el vector *sol* una posición a la izquierda e inicializando ,vo/|0] = 1.

La función factible? comprueba si la solución parcial calculada hasta el momento es extensible a una solución:

```
fun factible?(T : real, k : 1 ..5, último : I ..n, usados : 0..3, coste, suma-libres : real)
dev respuesta : bool
{ hay sellos para cubrir los huecos libres )
respuesta := (3 * (n — último) + 3 — usados) > (5 — k)
{ y se puede cubrir la cantidad pendiente )
respuesta := respuesta r\ suma-libres > (T — coste)
  ffun
```

El procedimiento principal que realiza la inicialización adecuada e imprime todos los franqueos admisibles es:

```
proc sellos2(e V[l..n] de real+. e T : real + )
```

```
var so/| 1 ..5] de nat
suma-todos := 0
para i = 1 hasta n hacer suma-todos
:= suma-todos + 3 * V[i| fpara
si 3 * n > 5 A suma-todos > T
entonces
último := I : usados := 0 ; coste := 0
sellos-va2( V, T. sol. I. último, usados,
coste, suma-todos)
fsi
  fproc
```

Apartado (c)----------------------------

El coste de las soluciones no depende del orden en el que aparecen los sellos, por lo que podemos lijar el orden de colocación y así no considerar soluciones repetidas (en términos de su coste). Por tanto modificaremos el algoritmo sellos-va2 del apartado anterior (donde no importa el orden de los sellos) para mantener la mejor solución encontrada hasta el momento.

Para calcular una estimación inferior de la mejor solución alcanzable desde el franqueo (parcial) actual rellenamos las casillas restantes con los sellos de menor valor aún no considerados. Esta estimación nunca puede ser menor que la tarifa a cubrir. Para calcular este valor nos interesa tener los sellos ordenados de menor a mayor valor Cuando se pasa al primer hijo, como se elige el sello de menor valor, la estimación coincide con la de su padre que ya era prometedor; como además *coste-mejor* no ha cambiado, el primer hijo siempre es prometedor y por consiguiente no hace falta comprobarlo.

Además, si se llega a alcanzar la tarifa *T* de forma exacta podremos terminar inmediatamente la búsqueda, porque tal solución es obviamente óptima.

El algoritmo es el siguiente:

I V[l| < V[2] < ... < V[«])

proc sellos-mín-va(e V[I ..n] de *real~.* **e** *T : real⁺. sol[* **1..5] de l..n.e¿ :**

1..5. *último* **: L.n.**
usados : 0..3, *coste, suma-libres ; real.*
sol-mejor[\..5] de *nat.*
coste-mejor : realy-. encontrada : bool)
(probamos un sello de tipo *último)*
si *usados* < 3 **entonces**
so/|Z:] := *último*
usados usados + 1 ; *coste := coste* + V[ií/rÍMio] { marcar)
suma-libres := suma-libres — V[ií/rimo]
si *k* = *5* **entonces**
si *coste* > *T* **entonces** { es solución)
sol-mejor := sol ; coste-mejor := coste
encontrada := (coste = T) { terminar}
fsi
si no
si factible?(T, *k. último, usados, coste, suma-libres)* **entonces**
sellos-mín-va(V. *T. sol. k* 4- 1. *último, usados, coste, suma-libres, sol-mejor, coste-mejor, encontrada)*
fsi
fsi
usados := usados — 1 : *coste := coste — V[último]* { desmarcar}
suma-libres := suma-libres + V[último]
fsi
{ descartamos los sellos no utilizados de tipo *último)*

suma-libres-act := suma-libres — (3 — *usados) * V [último]*
{ probamos con el resto de tipos de sellos }
usados-act := 1
s *:= último* + 1
mientras —•*encontrada /\ s < n* **hacer**
so/[it] := s
coste := coste + V[s] (marcar)
suma-libres-act := suma-libres-act — V[s] { quitamos un sello de tipo .v)
si coste-estimado(*V. T.k. s. usados-act, coste) < coste-mejor* entonces
si *k*: = 5 **entonces**
si *coste > T* **entonces** { es solución) *sol-mejor := sol; coste-mejor := coste encontrada := (coste — T)* { terminar)
fsi
si no
si factible?(T, *k. s. usados-act, coste, suma-libres-act)* **entonces**
sellos-mín-va(V, *T. sol. k* + 1, s, *usados-act. coste, suma-libres-act.*
sol-mejor, coste-mejor, encontrada)
fsi
fsi
fsi
coste ■. = *coste* — V[s] { desmarcar)
(quitamos los dos sellos restantes de tipo 5)
suma-libres-act := suma-libres-act—

2 V[s]*
s := s + I
fniientras
fproc
La función que calcula la estimación inferior es:
(V[J] < V[2] < ... < V[n])
fun coste-estimado(*V*[1 ,.n] **de** *real*$^+$*.*
T : real$^+$*. k :* 1..5. *último* : l..n,
usados : 0..3. *coste : rea!)* **dev**
estimación : real
estimación := coste
j := k + \ ; i ■*.= último*
mientras *j* < 5 A *i* < *n* **hacer**
si *usados* < 3 **entonces**
estimación := estimación + V[t]
usados := usados + 1 : *j := j* + 1
si no
i := i + 1 ; *usados :=* 0
fsi
fmientras
estimación := m¿x(estimación, T)
ffun

La función principal que devuelve la mejor solución es:
{ V[l| < V[2| < ... < V[n] }
fun sellos-mini V| 1 ../i] **de** *real* $^+$ *. T : real* $^+$ *)* **dcv (***sol-mejor[* 1. 5] **de** 1 *..n. coste-mejor : real^_)*
var ,w/| 1 .5] **de I** *n*
simia-todos := 0
para *i* = 1 **hasta** *n* **hacer** *suma-todos := simia-todos* + 3 * V[í] **fpara**
si 3 * *n > 5 A suma-todos > T*

entonces
último := 1 : *usados* := 0 ; *coste* := 0
coste-mejor := +oo : *encontrada* :=
falso
sellos-mín-va(V. *T. sol.* 1. *último,*
usados, coste, suma-todos, sol-mejor,
coste-mejor, encontrada **fsi**
fproc

-14.22. Empaquetamiento óptimo

Se tiene una colección de *n* objetos "moldeables" (cada uno con un volumen *vi*, con *i* entre 1 y *n*), que hay que empaquetar utilizando envases de capacidad *E*. Diseñar un algoritmo que calcule el empaquetamiento óptimo, es decir, que minimice la cantidad de envases utilizados, teniendo en cuenta que los objetos NO se pueden fraccionar.

--------------------------Solución--------------------------

Supondremos que todo objeto cabe en un envase vacío, es decir, $\forall i: 1 \leq i \leq n: v_i \leq E$. Ya que necesitamos como mucho *n* envases, podemos numerar los envases del 1 al *n*.

Las soluciones podemos representarlas en tuplas de la forma $(x_1,\ldots,x_n)$ siendo x_i el envase donde hemos colocado el objeto *i*. Como todos los envases vacíos son iguales, para cada objeto se puede usar uno de los envases ya ocupados, si cabe en alguno, o coger uno cualquiera sin usar (vacío). El primer objeto siempre se colocará en el primer envase (x_1 = 1).

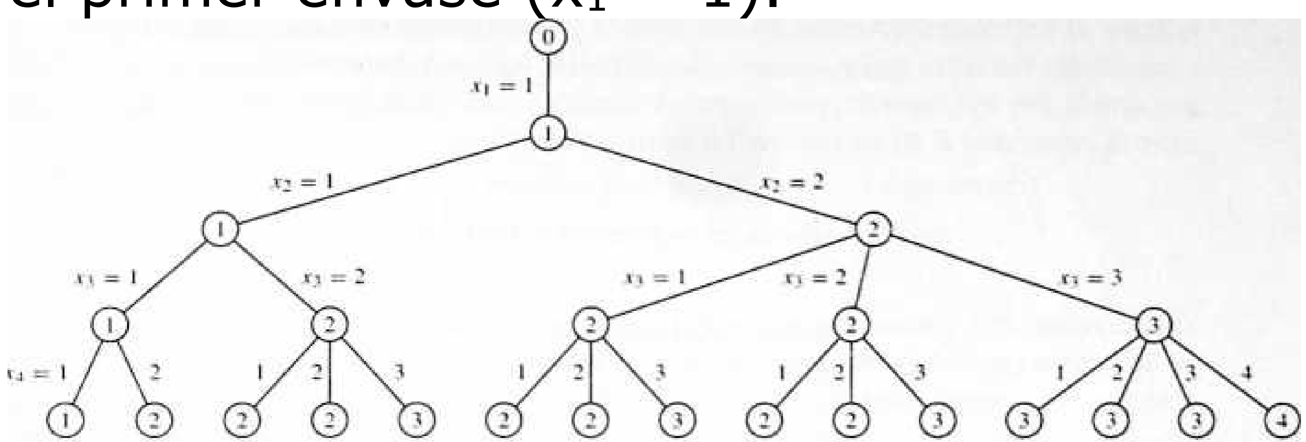


Figura 14.14: Árbol de exploración para el problema de los envases.

(Por ejemplo, para *n*=4, el árbol de exploración es el mostrado en la Figura

14.14, donde el número dibujado dentro de cada nodo indica el número total de envases utilizados hasta el momento).

Ya que para colocar un objeto en un envase tenemos que disponer de suficiente espacio disponible en el envase, llevaremos cuenta de la capacidad libre en cada envase mediante un vector *capacidad*[*1*..n].

El número de *envases* ya utilizados en una solución parcial es una posible estimación del número de envases necesarios al extender dicha solución. ¿Podemos ajustar más este valor? Una estimación correcta es considerar que los objetos restantes son fragmentadles, e intentar empaquetarlos todos comenzando por rellenar el volumen disponible en los envases ya utilizados. Para ello comenzamos por sumar todas las capacidades restantes en los envases ya utilizados, *hueco, y* restamos dicha cantidad del total de volumen que queda por empaquetar, *pendiente. A* continuación, basta dividir el volumen que sigue sin envasar entre la capacidad *E* de un envase. La estimación sería

$$\text{envasesEstimados} = \text{envases} + \text{máx}\left\{0, \left\lceil \frac{\text{pendiente} - \text{hueco}}{E} \right\rceil\right\}.$$

Sin embargo, esta estimación o bien es igual a *envases* cuando los objetos restantes caben en los envases ya utilizados (posiblemente partiéndose), o bien es igual a un valor que solamente

depende de todos los volúmenes *vi*, para *i* entre 1 *y n*, *y* no del reparto ya hecho en la solución parcial:

$$\text{óptimo} = \left\lceil \frac{\sum_{i=1}^{n} v_i}{E} \right\rceil,$$

y que es igual al número de envases en una solución óptima, por lo que este valor nunca sería mayor que el número de envases de la mejor solución encontrada hasta el momento y no ayudaría a podar.

Por tanto, utilizaremos simplemente *envases* como estimación y utilizaremos el valor *óptimo* para detener la búsqueda cuando encontremos una solución con dicho valor. La estimación para todos los hijos excepto el último (utilizar un envase nuevo) coincide con la del padre, por lo que solo comprobaremos si este último hijo es prometedor.

El algoritmo queda como sigue:

$$\left\{(\forall i: 1 \le i \le n: V[i] \le E) \wedge \text{ óptimo } = \left\lceil \left(\sum_{i=1}^{n} v_i\right)/E \right\rceil\right\}$$

proc **empaquetarVa**(**e** *E: real*⁺, **e** *V*[1..n] **de** *real*⁺, *sol*[1..n] de 1..*n*, **e** *k:* 1..*n*, *envases*: 1..*n*, *capacidad[1*..n] de *real*, **e** *óptimo:* 1 ..*n*, *solMejor[1..n]* **de** 1..n, *envasesMejor*: 1..n+1, *encontrada: bool)*

{probamos con cada envase ya utilizado}
i := 1;

```
mientras i ≤ envases ∧ ⌐encontrada hacer
    si capacidad[i] ≥ V[k] entonces
        sol[k] := i;
        capacidad[i] := capacidad[i] - V[k];  {marcar}
        si k=n entonces
            solMejor:= sol;
            envasesMejor:= envases;
            encontrada := (envasesMejor = óptimo);
            {terminar}
        si no
            empaquetarVa(E, V, sol, k+1, envases,
            capacidad, óptimo, solMejor, envasesMejor,
            encontrada);
        fsi
        capacidad[i] := capacidad[i] + V[k]; {desmarcar}
    fsi
    i := i + 1;
fmientras

si ⌐encontrada entonces
    {probamos con un envase nuevo}
    sol[k] := envases + 1;
    envases := envases + 1;
    capacidad[envases] := E - V[k]; {marcar}
    si envases < envasesMejor entonces
        si k=n entonces
            solMejor := sol;
            envasesMejor := envases;
            encontrada := (envasesMejor = óptimo)
            {terminar}
        si no
            empaquetarVa(E, V, sol, k+1, envases,
            capacidad, óptimo, solMejor, envasesMejor,
            encontrada);
        fsi
    fsi
    capacidad[envases] := E;
    envases := envases – 1; {desmarcar}
fsi
fproc
```

La función principal que realiza la inicialización adecuada es:

$\{\forall i : 1 \leq i \leq n : V[i] \leq E\}$

```
fun empaquetar(E: real⁺, V[1..n] de real⁺) dev
⟨solMejor[1..n] de 1..n, envasesMejor: 1..n+1⟩
var sol[1..n] de 1..n, capacidad[1..n] de real

sol[1] := 1;
envases : = 1;
capacidad[1] := E - V[1];
capacidad[2..n] := [E];
total := 0;
para i = 1 hasta n hacer
   total := total + V[i];
fpara
óptimo := ⌈ total/E ⌉;
encontrada := falso;
envasesMejor := n + 1; {peor que cualquier solución}
empaquetarVa(E, V, sol, 2, envases, capacidad,
óptimo, solMejor, envasesMejor, encontrada);
ffun
```

Capitulo 15

15. RAMIFICACIÓN Y PODA

CARACTERIZACION DE LOS PROBLEMAS

RAMIFICACIÓN Y PODA (*branch & bound* en inglés) es otra técnica de exploración de espacios de soluciones.

La diferencia esencial respecto a la técnica de *vuelta atrás* es la forma de recorrer el espacio de soluciones. Mientras en la *vuelta atrás* el recorrido se hacía en profundidad, mediante una pila (implícita) de nodos vivos, en ***ramificación y poda*** la gestión de los nodos vivos se realiza mediante una *cola con prioridad* (Capítulo 8), **expandiendo en cada momento el nodo vivo "más prometedor**". De este modo, aunque se puede usar *ramificación y poda* para buscar *una* o *todas* las soluciones posibles a un problema, en dichos casos no aporta nada nuevo con respecto a la resolución del problema por *vuelta atrás*, salvo un orden diferente al recorrer el árbol de búsqueda (*si bien esto puede resultar imprescindible para encontrar soluciones en problemas con un número no acotado de etapas, que pudieran dar lugar a ramas de profundidad no acotada y en las cuales la técnica de vuelta atrás podría no ser aplicable*). Por eso. la variante más utilizada de ramificación y poda es la

que **busca la solución óptima de entre todas las posibles** soluciones al problema, esperando encontrarla de forma más rápida que por *vuelta atrás*.

Recordemos las características de estos problemas de optimización:

- la solución es expresable en forma de tupla $(x_1,\ldots,x_n)$.
- es posible determinar si una tupla es una solución factible,
- es posible determinar si una tupla parcial $(x_1,\ldots,x_k)$ puede ser completada hasta una solución factible.

Además, es necesario disponer de una función costeEstimado (*mantendremos el convenio de hablar de "coste" cuando el problema sea de minimización y hablar de "beneficio" cuando el problema sea de maximización, pero el tratamiento en uno y otro caso es completamente simétrico: véase, por ejemplo, la solución al Ejercicio 15.1*) que, dada una tupla parcial $X = (x_1,\ldots,x_k)$ proporcione una cota inferior al coste de la mejor solución alcanzable desde *X*, es decir, de la mejor solución contenida en el subárbol que tiene por raíz *X*. Llamaremos costeReal(*X*) al coste de dicha mejor solución (normalmente este coste no se conocerá salvo recorriendo todo el subárbol). Entonces, ha de cumplirse que

costeEstimado(X) ≤ costeReal(X).

En el supuesto de que en el árbol de exploración las soluciones se encuentran únicamente en las hojas, suele cumplirse que costeEstimado(X) = costeReal(X) cuando X es solución.

Si costeEstimado(X) se aproxima suficientemente al valor del coste real, es de esperar que los nodos con un bajo coste estimado conducirán a soluciones con un bajo coste real, y por eso consideraremos como más prometedor el nodo vivo que tenga un coste estimado mínimo.

Además, como ya se indicó para la técnica de *vuelta atrás*, el coste estimado proporciona una forma adicional de poda. Al ser una cota inferior del coste alcanzable por cualquier solución derivable del nodo, si el coste estimado no es menor que el coste de la mejor solución obtenida hasta el momento, eso significa que no merece la pena seguir explorando esa rama del árbol, pues cualquier solución alcanzable va a ser peor que la que ya se tiene.

ESQUEMA DE RAMIFICACIÓN Y PODA DE COSTE MINIMO

Al igual que se hizo en el capítulo anterior para los esquemas de *vuelta atrás*, consideramos aquí árboles de exploración cuyos nodos solución solo se encuentran en las hojas.

El esquema general, para un problema de **minimización de costes**, se muestra a continuación:

```
fun ramificacionyPodaMín(T: árbol-de-
estados) dev ⟨solMejor: tupla, costeMejor: valor⟩
var X, Y : nodo, C : colapr[nodo]

Y:= raíz(T);
C:= cpVacía();
añadir(C, Y);
costeMejor:= +∞;
mientras ¬ esCpVacía?(C) ∧
costEstimado(mínimo(C)) < costeMejor hacer
  Y:= mínimo(C) ;
  eliminarMín(C);
  para todo hijo X de Y hacer
    si esSolución?(X) entonces
      si costeReal(X) < costeMejor entonces
        costeMejor := costeReal(X) ;
        solMejor := solución(X);
      fsi
    si no
      si esCompletable?(X) ∧ costeEstimado(X) <
      costeMejor entonces
        añadir(C, X);
      fsi
    fsi
  fpara
fmientras
ffun
```

En el esquema anterior, cada vez que extraemos el elemento mínimo de la cola con prioridad, determinamos si su coste estimado está por debajo del coste mejor obtenido hasta entonces y, en caso contrario, se termina la búsqueda, ya que los restantes nodos vivos también tendrán un coste estimado peor.

Otra posibilidad es mantener en la cola de nodos vivos solamente los que tengan oportunidad real de ser expandidos. Para ello, es necesario que cada vez que encontremos una solución mejor que la almacenada y, por tanto, modifiquemos

el valor de *costeMejor*, "depuremos" toda la cola de nodos vivos para eliminar aquellos cuyo coste estimado ha quedado ahora por encima (o igual) de *costeMejor*. De este modo, el algoritmo terminaría cuando la lista de nodos vivos quedase vacía. Este esquema alternativo es el siguiente:

```
fun ramificacionyPodaMín2(T: árboldeestados)
dev ⟨solMejor: tupla, costeMejor: valor⟩
var X, Y: nodo, C: colapr[nodo]

Y:= raíz(T);
C:= cpVacía() ;
añadir(C, Y);
costeMejor:= +∞;
mientras ¬esCpVacia?(C) hacer
  Y:= mínimo(C) ;
  eliminarMín(C);
  para todo hijo X de Y hacer
    si esSolución?(X) entonces
      si costeReal(X) < costeMejor entonces
        costeMejor:= costeReal(X);
        solMejor:= solución(X);
        depurar(C, costeMejor);
      fsi
    si no
      si esCompletable?(X) ∧ (costeEstimado(X) <
      coste-mejor) entonces
        añadir(C, X);
      fsi
    fsi
  fpara
fmientras
ffun
```

Si bien las soluciones que presentamos a continuación no realizan depuración alguna, la utilización de uno u otro esquema depende del problema particular a resolver y de la estructura que se utilice para implementar la cola con prioridad.

Las operaciones más frecuentes sobre la cola con prioridad de nodos vivos son la inserción de nuevos nodos y la extracción del elemento mínimo. Los montículos (véase el Ejercicio 8.4), por ejemplo, proporcionan una implementación eficiente de ambas operaciones (con un coste logarítmico respecto al tamaño de la cola) pero, en cambio, la depuración puede ser muy costosa. Sin embargo, si durante la exploración no se van a encontrar muchas soluciones, el coste mejor se mantendrá casi inalterable y la depuración se invocará con poca frecuencia. A cambio, la cola de nodos vivos se mantendrá con un tamaño mínimo, lo que facilitará la inserción de los nuevos nodos.

En cuanto a la información almacenada en cada nodo, en general incluye la solución parcial y la etapa correspondientes al estado del nodo, así como el valor estimado asociado, determinante de la prioridad para escoger el nodo vivo a expandir. Adicionalmente, y para facilitar el trabajo a desarrollar en cada nodo, se podrán almacenar marcadores (véase la Sección 14.2.2 en vuelta atrás).

ESQUEMA OPTIMISTA-PESIMISTA (MINIMIZACIÓN)

En los esquemas anteriores se mantiene en la variable *costeMejor* el coste asociado a la mejor solución calculada hasta el momento. Este valor se modifica sólo al encontrar una solución mejor, y al principio tiene el peor valor posible (+∞), lo que hace que la poda no sea efectiva hasta encontrar una solución y que después mejore solo lentamente.

Suponiendo que disponemos de una función que, dado un nodo X, calcule una *cota superior* del coste de la mejor solución alcanzable desde X, podría usarse para ir actualizando *costeMejor:* cada vez que encontremos un nodo factible con una cota superior *menor* que el valor actual de *costeMejor,* podremos actualizar *costeMejor* con el valor de dicha cota, lo que hará la poda más efectiva. Ahora *costeMejor* representa una cota superior del coste de la solución óptima (de coste mínimo) en el árbol completo. Usamos, por tanto, 2 cotas para cada nodo X:

costeOptimista(X) = cota inferior al coste de la mejor solución alcanzable desde X (antes costeEstimado(X)).

costePesimista(X) = cota superior del coste de la mejor solución alcanzable desde X.

Por tanto, dichas cotas cumplen:

costeOptimista(X) ≤ costeReal(X) ≤ costePesimista(X).

En el supuesto de que en el árbol de exploración las soluciones se encuentran únicamente en las hojas, suele cumplirse que

costeOptimista(X) = costeReal (X) = costePesimista(X)

cuando *X* es solución.

Además para cualquier solución *Y* alcanzable a partir de *X* se cumple que

costeOptimista(*X*) ≤ costeReal(*X*) ≤ coste(*Y*).

Sin embargo, *no* necesariamente se tiene que cumplir que

coste(Y) ≤ costePesimista(X).

Por tanto se puede utilizar como coste pesimista de un nodo X el coste de cualquier solución *Y* alcanzable a partir de X, aunque esta estimación puede ser poco aproximada (e incluso costosa de calcular).

El esquema general se muestra en el algoritmo siguiente:

```
fun rpOptPesMín(T: árboldeestados) dev
(solMejor: tupla, costeMejor: valor}
var X, Y: nodo, C: colapr[nodo]

Y:= raíz(T);
C:= cpVacía();
añadir(C, Y);
costeMejor:= costePesimista(Y);
mientras ¬ esCpVacía?(C) ∧
costeOptimista(mínimo(C)) ≤ costeMejor hacer
    Y:= mínimo(C);
    eliminarMín(C);
    para todo hijo X de Y hacer
        si esSolución?(X) entonces
            si costeReal(X) < costeMejor entonces
                costeMejor:= costeReal(X);
                solMejor:= X;
```

```
      fsi
    si no
      si esCompletable?(X) ∧ costeOptimista(X) ≤
      costeMejor entonces
        añadir(C, X);
        si costePesimista(X) < costeMejor
        entonces
          costeMejor := costePesimista(X);
        fsi
      fsi
    fsi
  fpara
fmientras
ffun
```

Nótese que para actualizar la mejor solución se utiliza la condición

costeReal(*X*) ≤ *costeMejor*

ya que ahora *costeMejor* es solo una cota superior del coste de la mejor solución, y no el coste real de una solución previamente encontrada.

Análogamente hemos relajado la restricción para mantener un nodo vivo a

costeOptimista(X) ≤ *costeMejor*

porque cuando costeOptimista(X) coincide con *costeMejor* debemos seguir expandiendo el nodo X ya que quizás la mejor solución esté en su subárbol.

Obsérvese que este esquema puede utilizarse solamente cuando sabemos si por debajo de un nodo X hay soluciones o no. Si pudiera no haberlas y no sabemos determinarlo, utilizar la estimación pesimista de X para actualizar *costeMejor* sería un error si realmente no hubiera soluciones por debajo de X.

EJERCICIOS RESUELTOS RYP

*Los ejercicios que se enuncian a continuación corresponden a los problemas de optimización que fueron propuestos en el CAPÍTULO ANTERIOR, dedicado a la técnica de **vuelta atrás**. Las soluciones que se proponen están basadas en las que fueron propuestas en dicho capítulo. En concreto, para cada problema se considerará el mismo árbol de exploración que entonces se definió. Además, en cada caso se utilizará como valor **optimista** el valor estimado utilizado para la vuelta atrás.*

15.1. Funcionarios

El Ministro de Desinformación y Decencia se ha propuesto hacer trabajar en firme a sus *n* funcionarios, para lo que se ha sacado de la manga *n* trabajos. A pesar de su ineficacia, todos los funcionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros y unos los hacen peor que otros. La información al respecto se recoge en dos tablas T[1..n,1..n] y E[1..n,1..n], donde T[*i*,*j*] representa el **tiempo** que el funcionario *i* tarda en realizar el trabajo *j* y E[*i*,*j*] representa la **eficacia** con la que el funcionario *i* realiza el trabajo *j*. Para justificar su puesto, el Ministro desea conocer la asignación óptima de trabajos a funcionarios en cada uno de los 2 sentidos diferentes (e independientes) siguientes:

(a) de modo que la suma total de tiempos sea mínima, y
(b) de modo que la suma total de

eficacias sea máxima.

-------------------------Solución--------------------------

Este ejercicio se resolvió mediante la técnica de *vuelta atrás* en el Ejercicio 14.15. Las soluciones (permutaciones de {1,....,*n*}) se representan mediante tuplas de la forma (x_1,…,x_n) donde x_i es el trabajo asignado al funcionario *i.*

Apartado (a)-----------------------

En este apartado el problema consiste en elegir los trabajos x_i tales que se minimice $\sum_{i=1}^{n} T[i, x_i]$.

En este caso utilizaremos las siguientes estimaciones para el esquema optimista-pesimista:

optimista: Se puede utilizar cualquiera de las estimaciones (cotas inferiores) presentadas en el Ejercicio 14.15 de *vuelta atrás*. En concreto utilizaremos aquella en que se lleva precalculado, para cada funcionario *k*,

$$opt[k] = \sum_{i=k+1}^{n} rápido[i]$$

donde *rápido[i]* = mín(T[i,j| | 1≤*j*≤n).

pesimista: Una posibilidad es derivar una solución cualquiera a partir de la solución parcial, asignando trabajos libres a los funcionarios restantes siguiendo el orden establecido. Disponiendo de un vector *asignado* que nos indique los trabajos que ya han sido asignados, este proceso

tiene coste lineal.
También se puede obtener una cota superior de cualquier solución alcanzable a partir de la solución parcial calculando el máximo global de la matriz *T*.

maxT = máx{T[i,*j*] | 1≤*i*≤*n* ∧ 1≤*j*≤n).

y aproximando superiormente el tiempo del resto de la solución con (n-*k*)*máxT*. Esta cota se puede mejorar calculando un máximo por cada fila:

lento[i] = máx{T[i,j] | 1≤*j*≤n).

De esta manera podemos precalcular la estimación del tiempo restante como

$$\sum_{i=k+1}^{n} lento[i]$$

Igual que hemos hecho para la cota optimista este valor se puede tener precalculado en un vector $pes[k] = \sum_{i=k+1}^{n} lento\,[i]$, y eso será lo que hagamos.

Podríamos afinar la estimación, más todavía, calculando *lento* dinámicamente entre los trabajos no repartidos ya a algún funcionario:

lento-din[i] = máx{T[i,*j*] | 1≤*j*≤*n* ∧ *j* no ha sido asignado}.

Pero esta cota superior puede no ser más ajustada que la correspondiente a obtener una solución cualquiera y además su cálculo tiene coste cuadrático.

El cálculo previo de las estimaciones se realiza mediante la siguiente función:

```
fun preCálculoEstimaciones(T[1..n, 1..n] de
```

```
real+) dev ⟨opt[0..n], pes[0..n] de real⟩
var rapido[1..n], lento[1..n] de real
 {cálculo de los mínimos y máximos por filas}
  para i = 1 hasta n hacer
    rapido[i] := T[i, 1];
    lento[i] := T[i, 1];
    para j = 2 hasta n hacer
      rapido[i] := mín(rapido[i], T[i,j]);
      lento[i] := máx(lento[i], T[i, j]);
    fpara
  fpara

 {cálculo de las estimaciones }
 opt[n] := 0 ;
 pes[n] := 0
  para i = n-1 hasta 0 paso - 1 hacer
    opt[i] := opt[i+1] + rapido[i+1];
    pes[i] := pes[i+1] + lento[i+1];
  fpara
ffun
```

$\{opt[k] = \sum_{i=k+1}^{n} \min_{1\leq j\leq n} \{T[i,j]\} \wedge pes[k] = \sum_{i=k+1}^{n} \text{máx}_{1\leq j\leq n}\{T[i,j]\}\}$

En cada nodo, además de la información usual (solución parcial, etapa y prioridad), guardamos también información sobre el tiempo acumulado y los trabajos que ya han sido asignados.

```
tipos
  nodo = reg
      sol[1..n] de 1 ..n
      k: 0..n
      tiempo : real
      tiempoOpt : real    {prioridad}
      asignado[1..n] de bool
    freg
ftipos
```

El algoritmo es el siguiente:

```
fun funcionariosMínRp(T[1..n, 1..n] de real+)
    dev ⟨solMejor[1..n] de 1..n, tiempoMejor:
    real⟩
var X, Y: nodo,
     C: colapr[nodo],
     opt[0..n], pes[0..n] de real

⟨opt,pes⟩ := preCálculoEstimaciones(T);
{generamos la raíz}
Y.k := 0 ;
Y.asignado[1..n] := [falso];
Y.tiempo := 0 ;
Y.tiempoOpt := opt[0];
C:= cpVacía();
añadir(C,Y)
tiempoMejor := pes[0];
 mientras ¬esCpVacía?(C) ∧
 mínimo(C).tiempoOpt ≤ tiempoMejor hacer
   Y:= mínimo(C) ;
   elimínarMín(C);
   {generamos los hijos de Y}
   X.k:= Y.k + 1;
   X.sol:= Y.sol;
   X.asignado:= Y.asignado;
   para t=1 hasta n hacer
    si ¬ X.asignado[t] entonces
     X.sol[X.k]:= t ;
     X.asignado[t]:= cierto;
     X.tiempo:= Y.tiempo + T[X.k, t];
     X.tiempoOpt:= X.tiempo + opt[X ,k];
     si X.tiempoOpt ≤ tiempoMejor
     entonces
      si X.k = n entonces
        sol-mejor := X.sol;
        tiempo-mejor := X.tiempo;
      si no
        añadir(C. X);
        tiempoMejor := mín(tiempoMejor,
        X.tiempo + pes[X.k]);
      fsi
     fsi
     X.asignado[t]:= falso
    fsi
   fpara
```

```
  fmientras
ffun
```

Apartado (b)-------------------------

Este apartado es dual al anterior donde en vez de minimizar tenemos que maximizar: el problema consiste en elegir los trabajos xi tales que se maximice $\sum_{i=1}^{n} E[i, x_i]$. Por tanto, el problema es de ***maximización*** y hay que modificar ligeramente los esquemas vistos en las Secciones 15.2 y 15.3. En concreto, la cola con prioridad será de máximos, y se cumplirá

benefOptimista(X) ≥ benefReal(X) ≥ benefPesimista(X).

En este caso utilizaremos las siguientes estimaciones para el esquema optimista-pesimista:

optimista: Se puede utilizar cualquiera de las estimaciones (cotas superiores) presentadas en el Ejercicio 14.15 de *vuelta atrás*. En concreto utilizaremos aquella en que se lleva precalculado para cada funcionario *k*

$$\text{opt}[k] = \sum_{i=k+1}^{n} \text{mejor}\ [i]$$

donde *mejor[i]* = máx(E[i, j] | 1≤*j*≤n).

pesimista: Como se trata de un problema de maximización y las eficacias asociadas son siempre positivas, la cota inferior más sencilla es la eficacia alcanzada hasta el momento, que para una solución parcial $(x_1,\dots x_k)$ es *eficacia* =

$\sum_{i=1}^{k} E[i, x_i]$. Pero cuando quedan muchos funcionarios por asignar, resulta muy poco ajustada. Análogamente al caso de minimización. podemos considerar la derivación de una solución cualquiera a partir de la solución parcial (coste lineal) o calcular el mínimo global de la matriz *E, (*o sea *minE),* y aproximar la eficacia del resto de la solución con (n-k) *mínE.*

También podemos calcular un mínimo por fila

peor[i] = mín{E[i, *j]* | 1≤j≤n)

y calcular la aproximación de la eficacia de la solución restante como

$$\text{pes}[k] = \sum_{i=k+1}^{n} \text{peor } [i]$$

Escogiendo esta última opción, podemos utilizar la misma función preCálculoEstimaciones que en el apartado anterior, pero asignando a *opt* los máximos y a *pes* los mínimos de la matriz *E.*

Análogamente al caso de minimización, en cada nodo, además de la información usual (solución parcial, etapa y prioridad), guardamos también información sobre la eficacia acumulada y los trabajos que ya han sido asignados.

tipos

nodo = reg

sol[1..n] **de** 1..n

k: 0..n

```
      eficacia: real
      eficaciaOpt: real      {prioridad}
      asignado[1..n] de bool
    freg
ftipos
```

```
fun funcionariosMaxRp(E[1..n, 1..n] de real⁺)
dev ⟨solMejor[1..n] de 1..n, eficaciaMejor: real⟩
var X, Y: nodo, C: colapr[nodo], opt[0..n],
pes[0..n] de real

⟨pes, opt⟩:= preCalculoEstimaciones(E);
{generamos la raíz}
Y.k:= 0;
Y.asignado[1..n]:= [falso];
Y.eficacia:= 0;
Y.eficaciaOpt:= opt[0];
C := cpVacía();
añadir(C, Y);
eficaciaMejor:= pes[0];
mientras ¬esCpVacía?(C) ∧
máximo(C).eficaciaOpt ≥ eficaciaMejor hacer
   Y:= máximo(C);
   eliminarMáx(C);
   {generamos los hijos de Y}
   X.k:= Y.k + 1;
   X.sol:= Y.sol;
   X.asignado:= Y.asignado
   para t=1 hasta n hacer
      si ¬X.asignado[t] entonces
         X.sol[X.k]:= t ;
         X.asignado[t]:= cierto;
         X.eficacia:= Y.eficacia + E[X.k, t];
         X.eficaciaOpt:= X.eficacia + opt[X.k];
         si X.eficaciaOpt ≥ eficaciaMejor entonces
            si X.k = n entonces
               solMejor:= X.sol;
               eficaciaMejor:= X.eficacia;
            si no
               añadir(C, X);
               eficaciaMejor:= máx(eficaciaMejor,
               X.eficacia + pes[X.k]);
```

```
fsi
fsi
X.asignado[t]:= falso;
fsi
fpara
fmientras
ffun
```

15.2. Mochila Ali Baba

Cuando Alí-Babá entra en la Cueva de los 40 Ladrones encuentra allí objetos valiosos. Alí-Babá conoce el peso y valor (numeros reales) de cada uno de los objetos en la cueva. Solo puede llevar consigo las riquezas que quepan en su **MOCHILA**, que soporta un peso máximo conocido. Determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

------------------------Solución---------------------------

Suponiendo que en la cueva hay *n* objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$, para i entre 1 y *n, y* considerando que la mochila soporta un peso total máximo $M > 0$. el problema consiste en maximizar $\sum_{i=1}^{n} x_i v_i$ con la restricción $\sum_{i=1}^{n} x_i p_i \leq M$ donde $x_i \in \{0, 1)$ indica si hemos cogido (1) o no (0) el objeto *i.*

En cuanto a las estimaciones a utilizar para el esquema optimista-pesimista, tenemos:

optimista: Como ya se vio para la solución de *vuelta atrás* (*Ejercicio 14.19*), se consigue una cota superior del beneficio a obtener si se consideran los objetos fraccionables y se aplica el algoritmo voraz visto en el *Ejercicio 12.5*. Para utilizar este algoritmo necesitamos que los objetos estén ordenados de mayor a menor valor por unidad de peso, v_i/p_i.

pesimista: Podríamos tomar como cota

inferior el valor de los objetos que ya se han cogido, pero una cota que se aproxima más al beneficio real consiste en probar una posible solución: de entre los objetos sobre los cuales todavía no se ha decidido, se incorporan a la mochila todos los objetos que se pueda, considerándolos en el orden establecido. Cuando nos encontramos en un nodo en el que el último objeto considerado se ha metido en la mochila, la cota pesimista coincide con la de su padre por lo que no se podrá mejorar la variable *beneficio-mejor.*

En cada nodo, además de la información usual (solución parcial, etapa y prioridad), guardamos también el peso y el beneficio acumulados.

```
tipos
  nodo = reg
        sol[1..n] de 0..1
        k: 0..n
        peso, beneficio: real
        beneficioOpt: real    {prioridad}
      freg
ftipos
```

Obsérvese que las estimaciones de un hijo izquierdo coinciden con las de su padre. Además, cuando se generan los hijos se acaba de comprobar que el padre sigue siendo prometedor. Esto hace que un hijo izquierdo *X* factible sea siempre prometedor (*X.beneficioOpt* ≥ *beneficioMejor*).

El algoritmo es el siguiente:

$$\left\{\frac{V[1]}{P[1]} \geq \frac{V[2]}{P[2]} \geq \cdots \geq \frac{V[n]}{P[n]}\right\}$$

```
fun mochilaRp(P[1..n], V[1..n] de real+, M:
real+) dev ⟨solMejor[1..n] de 0..1,
beneficioMejor: real⟩
var X, Y: nodo, C: colapr[nodo]

{generamos la raíz}
Y.k:= 0;
Y.peso:= 0;
Y.beneficio:= 0;
⟨Y.beneficioOpt, beneficioMejor⟩ :=
cálculoEstimaciones(P, V, M, Y.k, Y.peso,
Y.beneficio);
C:= cpVacía();
añadir(C, Y);
mientras ¬esCpVacía?(C) ∧
maximo(C).beneficioOpt > beneficioMejor hacer
   Y:= máximo(C);
   eliminarMáx(C);
   X.k:= Y.k + 1;
   X.sol:= Y.sol;
   {probamos a meter el objeto en la mochila}
   si Y.peso + P[X.k] ≤ M entonces
      {es factible y, por tanto, las estimaciones coinciden
      con las de Y}
      {beneficioOpt(X) = beneficioOpt(Y) ≥
      beneficioMejor}
      X.sol[X.k]:= 1;
      X.peso:= Y.peso + P[X.k];
      X.beneficio:= Y.beneficio + V[X.k];
      X.beneficioOpt:= Y.beneficioOpt;
      si X.k = n entonces
         {beneficio(X) = beneficioOpt(X) ≥
         beneficioMejor}
            solMejor:= X.sol;
            beneficioMejor:= X.beneficio;
      si no
            añadir(C, X);
            {no se puede mejorar beneficioMejor}
      fsi
   fsi

   {probamos a no meter el objeto (siempre es factible)}
   ⟨X.beneficioOpt, pes⟩:= cálculoEstimaciones(P,
   V, M, X.k, Y.peso, Y.beneficio);
```

```
        si X.beneficioOpt ≥ beneficioMejor entonces
          X.sol[X.k]:=0;
          X.peso:= Y.peso;
          X.beneficio:= Y.beneficio;
          si X.k = n entonces
            solMejor:= X.sol;
            beneficioMejor:= X.beneficio;
          si no
            añadir(C, X);
            beneficioMejor:= máx(beneficioMejor, pes)
          fsi
        fsi
      fmientras
    ffun
```

La siguiente función calcula las estimaciones optimista y pesimista, considerando que los objetos están ordenados de la forma $\frac{v_i}{p_i} \geq \frac{v_{i+1}}{p_{i+1}}$, según el método voraz del Ejercicio 12.5.

$$\left\{\frac{V[1]}{P[1]} \geq \frac{V[2]}{P[2]} \geq \cdots \geq \frac{V[n]}{P[n]}\right\}$$

```
fun cálculoEstimaciones(P[1..n], V[1..n]
de real⁺, M: real⁺, k: 0..n, peso, beneficio:
real) dev ⟨opt, pes: real⟩

hueco:= M – peso;
pes:= beneficio;
opt := beneficio;
j:=k+1;
mientras j≤n ∧ P[j]≤hueco hacer
   {podemos coger el objeto j entero}
   hueco:=hueco - P[j];
   opt:= opt + V[j];
   pes:= pes + V[j];
   j:= j + 1;
fmientras

si j≤n entonces    {quedan objetos por probar}
   {fraccionamos el objeto j (solución voraz)}
   opt:= opt + (hueco/P[j]) * V[j];
   {extendemos a una solución en la versión 0/1}
   j:= j+1;
```

```
mientras j≤n ∧ hueco>0 hacer
    si P[j] ≤ hueco entonces
        hueco:=hueco - P[j];
        pes:= pes + V[j];
    fsi
    j:= j + 1;
fmientras
fsi
ffun
```

15.3. Cinta canciones

Pepe Casanova es un ligón de los de antaño, que intenta encandilar a las chicas con canciones románticas. A tal efecto, y de cara al veraneo en una playa del sur, decide conseguir una cinta para el radiocasete de su coche con las mejores canciones de amor. Pepe es muy peculiar en sus gustos, y además anda algo escaso de dinero, por lo que en lugar de comprar una de tantas recopilaciones que circulan por el mercado discográfico, quiere grabársela él mismo. Rebuscando entre sus viejos vinilos ha confeccionado la lista de sus *n* canciones favoritas, junto con la duración individual de cada una. Lamentablemente, su cinta (de 2 caras) de *T* minutos no tiene capacidad suficiente para contener todas las canciones, así que Pepe ha otorgado una puntuación a cada canción (cuanto más le gusta, mayor es la puntuación). Ayudar a Pepe a conseguir la mejor cinta, teniendo en cuenta que las canciones escogidas han de caber enteras y no es admisible que una canción se corte a la mitad al final de una cara de la cinta.

--------------------------Solución--------------------------

Este problema es muy parecido al de la **MOCHILA** de Ramificacion y Poda (Ejercicio 15.2). Recordemos que en la solución de *vuelta atrás* (*Ejercicio 14.20*) se consideraba en cada etapa una canción y se consideraba grabarla en cada una de las 2 caras (siempre que

hubiera espacio suficiente) así como la posibilidad de no grabarla en la cinta. Además, la posibilidad de grabar en la segunda cara solo se considera cuando la ocupación de las 2 caras es distinta, ya que si no estaríamos considerando casos simétricos.

En cuanto a las estimaciones a utilizar para aplicar el esquema optimista-pesimista, tenemos:

- **optimista**: Una cota superior adecuada es el beneficio estimado visto para la solución de *vuelta atrás*, es decir, considerar como espacio libre total la suma del espacio libre en cada cara y grabar el máximo posible de canciones, aunque alguna quede cortada. Para esto utilizaremos el método voraz del problema de la mochila visto en el Ejercicio 12.5. y por tanto es necesario tener ordenadas las canciones en orden decreciente de puntuación por unidad de tiempo. *pi/di.*
- **pesimista**: Como cota inferior, intentaremos ir grabando en orden las canciones aún no grabadas, en la primera cara y, si no caben, en la segunda. Es decir, extendemos la solución parcial a una posible solución.

Al igual que ocurría en la solución de *vuelta atrás*, al generar el primer hijo de un nodo, el beneficio optimista coincide con el del padre, del que se acaba de comprobar que sigue siendo prometedor,

por lo que no hay que repetir el test que pregunta si interesa expandir. En el segundo hijo, el beneficio optimista coincide con el del padre, y aunque el beneficio mejor puede haber cambiado al generar el primer hijo, no puede haberse hecho mayor que el optimista. Por tanto, y al ser aquí las comparaciones no estrictas, no es necesario volver a hacer el test. Para el tercer hijo la cota sí puede cambiar, por lo que se realiza de nuevo el test.

En cada nodo, además de la información usual (solución parcial, etapa y prioridad), guardamos también la ocupación en cada cara de la cinta y el beneficio acumulado.

tipos
 nodo = **reg**
  sol[1..n] de 0..2
  k: 0..n
  beneficio: real
  beneficioOpt: real {prioridad}
  ocupada[1..2] **de** *real*
 freg
ftipos

El algoritmo es el siguiente:

$$\left\{\frac{P[1]}{D[1]} \geq \cdots \geq \frac{P[n]}{D[n]}\right\}$$

fun cancionesRp(*D*[1..n], *P*[1..n] **de** *real*⁺, *T*: *real*⁺) **dev** ⟨*solMejor*[1..n] **de** 0..2, *beneficioMejor*: *real*⟩
var *X, Y*: *nodo*, *C*: *colapr[nodo]*

```
{generamos la raíz}
Y.k := 0;
Y.ocupada[1..2] := [0];
Y.beneficio := 0;
Y.beneficioOpt := cálculoOptimista(D, P, T, Y.k,
Y.beneficio, Y.ocupada);
C := cpVacía();
añadir(C, Y);
beneficioMejor := cálculoPesimista(D, P, T, Y.k,
Y.beneficio, Y.ocupada);
mientras ¬esCpVacía?(C) ∧
máximo(C).beneficioOpt ≥ beneficioMejor hacer
  Y := máximo(C);
  eliminarMáx(C);
  X.k := Y.k + 1;
  X.sol := Y.sol;
  {grabar la canción X.k en la cara 1}
  si Y.ocupada[1] + D[X.k] ≤ T/2 entonces
  {beneficioOpt(X) = beneficioOpt(Y) ≥
  beneficioMejor}
    X.sol[X.k] := 1;
    X.ocupada[1] := Y.ocupada[1] + D[X.k];
    X.ocupada[2] := Y.ocupada[2];
    X.beneficio := Y.beneficio + P[X.k];
    X.beneficioOpt := Y.beneficioOpt;
    si X.k = n entonces
      solMejor := X.sol;
      beneficioMejor := X.beneficio;
    si no
      añadir(C, X);
      {la estimación pesimista también coincide y no
      puede mejorar beneficioMejor}
    fsi
  fsi

  si Y.ocupada[1] ≠ Y.ocupada[2] entonces
  {grabar la canción X.k en la cara 2}
    {la estimación optimista coincide con la de Y, y no
    es menor que beneficio-mejor}
    si Y.ocupada[2] + D[X.k] ≤ T/2 entonces
      X.sol[X.k] := 2
      X.ocupada[1] := Y.ocupada[1];
      X.ocupada[2] := Y.ocupada[2] + D[X.k];
```

```
X.beneficio := Y.beneficio + P[X.k];
X.beneficioOpt := Y.beneficioOpt;
si X.k = n entonces
    solMejor := X.sol;
    beneficioMejor := X.beneficio;
si no
    añadir(C, X);
    {la estimación pesimista no coincide con la de Y}
    pes := cálculoPesimista(D, P, T, X.k,
    X.beneficio, X.ocupada);
    beneficioMejor := máx(beneficioMejor,
    pes);
fsi
fsi
fsi

{no grabar la canción k (siempre es factible)}
X.ocupada := Y.ocupada;
X.beneficio := Y.beneficio;
X.beneficioOpt := cálculoOptimista(D, P, T,
X.k, X.beneficio, X.ocupada);
si X.beneficioOpt ≥ beneficioMejor entonces
    X.sol[X.k] := 0;
    si X.k = n entonces
        solMejor := X.sol;
        beneficioMejor := X.beneficio;
    si no
        añadir(C, X)
        pes := cálculoPesimista(D, P, T, X.k,
        X.beneficio, X.ocupada);
        beneficioMejor := máx(beneficioMejor, pes);
    fsi
fsi
fmientras
ffun
```

Las 2 funciones siguientes sirven para obtener las cotas deseadas:

$$\left\{\frac{P[1]}{D[1]} \geq \cdots \geq \frac{P[n]}{D[n]}\right\}$$

```
fun cálculoOptimista(D[1..n], P[1..n] de real⁺, T:
real⁺, k: 0..n, beneficio: real, ocupada[1..2] de
real) dev opt: real
```

```
hueco := T - (ocupada[1] + ocupada[2])
opt := beneficio;
j := k + 1;
mientras j ≤ n ∧ D[j] ≤ hueco hacer
   {podemos grabar la canción j entera}
   hueco := hueco - D[j];
   opt := opt + P[j];
   j := j + 1;
fmientras

si j ≤ n entonces {quedan canciones por probar}
   {cortamos la canción j};
   opt := opt + (hueco/ D[j]) * P[j];
fsi
ffun
```

```
fun cálculoPesimista(D[1..n], P[1..n] de real⁺, T:
real⁺, k: 0..n, beneficio: real, ocupada[1..2] de
real) dev pes: real
var ocupadaAux[1..2] de real

ocupadaAux := ocupada;
pes := beneficio;
j := k+1;
mientras j≤n ∧ (ocupadaAux[1] ≤ T/2 ∨
ocupadaAux[2] ≤ T/2) hacer
   si ocupadaAux[1] + D[j] ≤ T/2 entonces
      ocupadaAux[1] := ocupadaAux[1] + D[j];
      pes := pes + P[j];
   si no
      si ocupadaAux[2] + D[j] ≤ T/2 entonces
         ocupadaAux[2] := ocupadaAux[2] + D[j];
         pes := pes + P[j];
      fsi
   fsi
   j := j+1;
fmientras
ffun
```

15.4. Huertas

El tío Facundo posee *n* huertas, cada una con un tipo diferente de árboles frutales. Las frutas ya han madurado y **es** hora de recolectarlas. El tío Facundo conoce, para cada una de las huertas, el beneficio *bi* que obtendría por la venta de lo recolectado. El tiempo que se tarda en recoger los frutos de cada finca es asimismo variable **y** viene dado por También sabe los días *di* que tardan en pudrirse los frutos de cada huerta**.** Ayudar al tío Facundo a decidir qué debe recolectar para maximizar el beneficio total obtenido**.**

-----------------Solución--------------------

En el Ejercicio 14.16 vimos la solución a este problema mediante la técnica de vuelta atrás. Representamos las soluciones mediante tuplas $(x_1,\ldots,x_n)$ donde $x_i=1$ indica que los frutos de la huerta *i* se recolectan mientras que $x_i=0$ indica que los frutos de la huerta *i* no se recolectan.

También vimos cómo comprobar que un subconjunto de huertas es factible, es decir, que todas pueden recolectarse sin superar su plazo, manteniendo las huertas ordenadas por tiempo de caducidad creciente.

En cuanto a las estimaciones a utilizar para el esquema optimista-pesimista, tenemos:

optimista: Como ya vimos en el Ejercicio 14.16. podemos aproximar

superiormente el beneficio obtenible sumando el beneficio de todas las huertas que pueden recolectarse sin llegar a su fecha de caducidad después de las ya elegidas (es decir, empezando a recolectar cada una justo después de terminar con la última finca *ya* elegida).

pesimista: Igual que en el Ejercicio 15.2. podemos obtener una cota pesimista calculando una posible solución extensión de la que tengamos: las huertas no consideradas se van recolectando en orden (acumulando el tiempo), siempre que no se supere su fecha de caducidad.

En cada nodo, además de la información usual (solución parcial, etapa y prioridad), mantendremos el tiempo y beneficio acumulados.

```
tipos
  nodo = reg
        sol[ 1 ..n] de 0.. 1
        k : 0..n
        tiempo, beneficio : real
        beneficioOpt : real   {prioridad}
     freg
ftipos
```

El algoritmo es el siguiente:

{D[1]≤ … ≤D[n]}

```
fun huertasRp(T[1..n], D[1..n), B[1..n] de
real+) dev ⟨solMejor[1..n] de 0..1,
beneficioMejor: real⟩
var X, Y: nodo, C: colapr[nodo]
```

```
{generamos la raíz}
Y.k := 0;
Y.tiempo := 0;
Y.beneficio := 0;
Y.beneficioOpt := cálculoOptimista(T, D, B, Y.k,
Y.tiempo, Y.beneficio)
C := cpVacía();
añadir(C, Y);
beneficioMejor := cálculoPesimista(T, D, B, Y.k.
Y.tiempo, Y.beneficio);
mientras ¬esCpVacía?(C) ∧
(máximo(C).beneficioOpt ≥ beneficioMejor)
hacer
    Y := máximo(C);
    eliminarMáx(C);
    X.k := Y.k + 1;
    X.sol := Y.sol;

    {hijo izquierdo — recolectar}
    si Y.tiempo + T[X.k] ≤ D[X.k] entonces {es
    factible}
        X.sol[X.k] := 1;
        X.tiempo := Y.tiempo + T[X.k];
        X.beneficio := Y.beneficio + B[X.k];
        X.beneficioOpt := cálculoOptimista(T, D, B, X.k,
        X.tiempo, X.beneficio);
        si X.beneficioOpt ≥ beneficioMejor entonces
            si X.k = n entonces
                solMejor := X.sol;
                beneficioMejor := X.beneficio;
            si no
                añadir(C, X);
                { la estimación pesimista coincide con la de Y }
                { beneficioMejor no cambia}
            fsi
        fsi
    fsi

    {hijo derecho — no recolectar}
    X.sol[X.k] := 0;
    X.tiempo := Y.tiempo;
    X.beneficio := Y.beneficio;
    X.beneficioOpt := cálculoOptimista(T, D, B, X.k,
    X.tiempo, X.beneficio);
    si X.beneficioOpt ≥ beneficioMejor entonces
        si X.k = n entonces
```

```
      solMejor := X.sol;
      beneficioMejor := X.beneficio;
    si no
      añadir(C, X)
      pes := cálculoPesimista(T, D, B, X.k,
      X.tiempo, X.beneficio);
      beneficioMejor := máx(beneficioMejor, pes)
    fsi
  fsi
 fmientras
ffun
```

Las funciones que calculan las estimaciones son las siguientes:

```
fun cálculo-optimista(T[1..n], D[1..n], B[l..n] de
real+, k: 0..n, tiempo, beneficio : real) dev opt :
real

opt := beneficio;
 para i = k+1 hasta n hacer
   si tiempo + T[i] ≤ D[i] entonces
     opt := opt + B[i];
   fsi
 fpara
ffun
```

```
fun cálculoPesimista(T[1..n], D[1..n], B[1..n] de
real+, k: 0..n, tiempo, beneficio: real) dev pes:
real

pes := beneficio;
tiempoPes := tiempo;
 para i = k+1 hasta n hacer
   si tiempoPes + T[i] ≤ D[i] entonces
     pes := pes + B[i];
     tiempoPes := tiempoPes + T[i];
   fsi
 fpara
ffun
```

15.5. Componentes electronicos

Tenemos un conjunto de *n* componentes electrónicos para colocar en *n* posiciones sobre una placa. Se nos dan 2 matrices N y D, ambas de dimensiones *n* x *n*, donde *N[i, j]* indica el número de conexiones necesarias entre la componente *i* y la componente *j*, *y D[p, q]* indica la distancia sobre la placa entre la posición *p y* la posición *q* (ambas matrices son simétricas y con diagonales nulas).

Un cableado $(x_1,\ldots,x_n)$ de la placa consiste en la colocación de cada componente *i* en una posición *xi*. La longitud total de este cableado viene dada por la fórmula:

$$\sum_{i<j} N[i,j]D\left[x_i,x_j\right].$$

Escribir un algoritmo para encontrar el cableado de longitud mínima.

-----------------Solución----------------

En el Ejercicio 14.17 se vio que las soluciones (permutaciones de {1,…*n*}) son los posibles cableados $(x_1,\ldots,x_n)$ donde *xi* es la posición asignada a la componente *i*.

Vamos a utilizar las siguientes estimaciones:

optimista: En el Ejercicio 14.17 vimos cómo calcular una cota inferior del coste de la mejor solución a partir del mínimo global de la matriz de distancias *D*. Así, para una solución parcial $(x_1,\ldots,x_k)$ con coste acumulado

coste, tenemos:

$$\text{coste-opt} = \text{coste} + \min D \sum_{j=k+1}^{n} \sum_{i=1}^{j-1} N[i,j].$$

pesimista: Aplicando el mismo razonamiento calculamos una cota superior a partir del máximo global:

$$\text{coste-pes} = \text{coste} + \text{máx}\, D \sum_{j=k+1}^{n} \sum_{i=1}^{j-1} N[i,j].$$

En ambos casos las sumas se pueden precalcular ya que solo dependen del valor de *k* y de la matriz *N.*

En cada nodo, además de la información usual (solución parcial, etapa y prioridad), llevamos cuenta del coste acumulado y de las posiciones que ya se han usado:

```
tipos
      nodo = reg
              sol[1..n] de 1..n
              k: 0..n
              coste: real
              coste_opt: real   {prioridad}
              usada[1..n] de bool
           freg
ftipos
```

El algoritmo resultante es el siguiente:

```
fun cableadoRp(N[1..n, 1..n] de nat, D[1..n, 1
..n] de real) dev ⟨solMejor[1..n] de 1..n,
costeMejor: real⟩
var X, Y: nodo, C: colapr[nodo], opt[0..n],
pes[0..n] de real
```

```
⟨opt, pes⟩ := preCálculoEstimaciones(W, D)
{generamos la raíz}
Y.k := 0 ;
Y.usada[1..n] := [falso]
Y.coste := 0;
Y.costeOpt := opt[0]
C := cpVacía();
añadir(C, Y);
costeMejor := pes[0];
mientras ¬esCpVacía?(C) ∧
mínimo(C).costeOpt < costeMejor hacer
  Y:= mínimo(C);
  eliminarMín(C);
  {generamos los hijos de Y}
  X.k := Y.k + 1;
  X.sol := Y.sol;
  X.usada := Y.usada;
  para p=1 hasta n hacer
    si ¬X.usada[p] entonces
      X.sol[X.k] := p;
      X.usada[p] := cierto;
      X.coste := Y.coste + costeComponente(N, D,
      X.sol, X.k);
      X.costeOpt := X.coste + opt[X.k];
      si X.costeOpt ≤ costeMejor entonces
        si X.k = n entonces
          solMejor := X.sol;
          costeMejor := X.coste;
        si no
          añadir(C, X);
          costeMejor := min(costeMejor, X.coste +
          pes[X.k])
        fsi
      fsi
      X.usada[p] := falso;
    fsi
  fpara
fmientras
ffun
```

Para calcular el coste de añadir una nueva componente a la solución parcial se utiliza la función coste-componente implementada en el Ejercicio 14.17.

La función necesaria para el cálculo previo de las estimaciones es:

```
fun preCálculoEstimaciones(N[1..n, 1..n] de nat,
D[1..n, 1..n] de real)
dev ⟨opt[0..n], pes[0..n] de real⟩

{cálculo del mínimo y del máximo en D}
mínD := +∞ ;
máxD := 0;
para i=1 hasta n hacer
  para j= i+1 hasta n hacer
    mínD := mín(mínD, D[i,j]);
    máxD := máx(máxD, D[i,j]);
  fpara
fpara

{cálculo de las estimaciones}
opt[n] := 0;
para k= n-1 hasta 0 paso - 1 hacer
  opt[k]:= opt[k+1];
  para i=1 hasta k hacer
    opt[k]:= opt[k] + N[i,k+1];
  fpara
fpara

para k=0 hasta n hacer
  pes[k]:= máxD * opt[k];
  opt[k]:= mínD * opt[k}];
fpara
ffun
```

$$\left\{ \mathrm{opt}[k] = minD \sum_{j=k+1}^{n} \sum_{i=1}^{j-1} N[i,j] \wedge pes[k] \right.$$

$$\left. = \text{máx}\, D \sum_{j=k+1}^{n} \sum_{i=1}^{j-1} N[i,j] \right\}$$

15.6. Parejas afines

Un grupo de amigos, formado por *n* parejas (n mujeres y *n* hombres), se reúnen para cenar. El protocolo del buen comensal exige que, a la hora de sentarse a la mesa (redonda), hombres y mujeres deben alternarse y que nadie debe sentarse al lado de su pareja habitual. Para que la velada sea lo más agradable posible**,** hay que maximizar el grado de bienestar total, obtenido sumando los grados de afinidad mutuos entre los comensales sentados en posiciones adyacentes. Al efecto, se dispone de sendas matrices **que** nos indican la afinidad entre hombres y mujeres y entre mujeres y hombres. Una vez establecida cada pareja de vecinos, la afinidad mutua se calcula multiplicando la de cada miembro por el **contrario.** Diseñar **un** algoritmo que encuentre una solución óptima al problema.

-----------------Solución-----------------

Recordemos que en el Ejercicio 14.18 fijamos que en las posiciones impares se sentarían hombres y en las pares mujeres, y representamos las soluciones como tuplas $(x_1,\dots,x_{2n})$ donde xi=*j* indica que en la silla *i* se sienta el hombre de la pareja *j* si *i* es impar, o la mujer de la pareja *j* si i es par. Para evitar soluciones repetidas, fijamos que en la primera silla se sienta el hombre de la primera pareja, x1=1.

En cuanto a las cotas, en el Ejercicio

14.18 se calculaba una cota superior (optimista) de la preferencia alcanzable desde una solución parcial a partir de la máxima preferencia de cualquier hombre por cualquier mujer *(máxH)* y de cualquier mujer por cualquier hombre *(máxM).*

Como a partir de una solución parcial dada, puede ocurrir que no exista ninguna solución que cumpla los requisitos exigidos, ***no*** se puede utilizar el esquema optimista-pesimista, y usaremos la función de estimación previamente explicada para dar prioridad a los nodos vivos.

En cada nodo guardamos la solución parcial, la etapa, el beneficio acumulado y estimado (prioridad), y los hombres y mujeres ya sentados:

```
tipos
  nodo = reg
    sol[1..2n] de nat
    k: 1..2n
    bienestar: real
    bienestarEstimado: real //{prioridad}
    sentado[0..1, 1..n] de bool
  freg
ftipos
```

El algoritmo es el siguiente:

```
fun cenaRp(P[0..1, 1..n, 1..n] de real) dev
⟨solMejor[1..2n] de nat, bienestarMejor: real∞⟩ var X, Y:
nodo, C: colapr[nodo]

//{cálculo de los máximos}
máxH := -∞;
máxM := - ∞;
para i = 1 hasta n hacer
  para j = 1 hasta n hacer
    si i ≠ j entonces
```

```
    máxH := máx(máxH, P[1, i, j]);
    máxM := máx(máxM, P[0, i, j]);
   fsi
  fpara
 fpara

//{generamos la raíz: en la primera silla se sienta el hombre de
la primera pareja}
Y.k := 1;
y.sol[l] := 1;
Y.bienestar := 0;
Y.bienestarEstimado := (2 * n) * (máxH * máxM)
Y.sentado[0.. 1, 1..n] := [falso];
Y.sentado[1, 1] := cierto;
C := cpVacía();
añadir(C, Y);
bienestarMejor := -∞
mientras ⌐esCpVacía?(C) ∧
máximo(C).bienestarEstimado > bienestarMejor hacer
Y := máximo(C) ; eliminarMáx(C);
//{generamos los hijos de Y}
X.k := Y.k + 1; X.sol := Y.sol; X.sentado := Y.sentado;
sexo := X.k mod 2; anterior := X.sol[X.k -1];
para i = 1 hasta n hacer
si ⌐X.sentado[sexo.i] ∧ i ≠ anterior entonces
X.sol[X.k] := i; X.sentado[sexo, i] := cierto;
X.bienestar := Y.bienestar + P[sexo, i, anterior] * P[1-
sexo, anterior, i];
si X.k = 2 * n entonces
si i ≠ 1 ∧ X.bienestar + P[1, 1, i] * P[0, i, 1] >
bienestarMejor entonces
solMejor := X.sol ;
bienestarMejor := X.bienestar + P[1, 1,i]* P[0,i, 1];
fsi
si no
X.bienestarEstimado : = X.bienestar + (2 * n - X.k + 1)
* máxH * máxM;
si X.bienestarEstimado > bienestarMejor entonces
añadir(C, X);
fsi
fsi
X.sentado[sexo, i] := falso;
fsi
fpara
fmientras
ffun
```

15.7. Franquear 5 sellos

Juanito tiene un lote de sellos de *n* valores diferentes, disponiendo de *3* sellos de cada valor. Conoce las diferentes tarifas de franqueo de cartas postales, y sabe que el franqueo de una carta lo componen 5 y sólo 5 sellos, de forma que un franqueo solo es admisible si se alcanza la tarifa correspondiente y se usan exactamente 5 sellos. Escribir un algoritmo para obtener una solución que franquee, de forma admisible, una postal de tarifa *T* con el mínimo coste posible.

----------------Solución-----------------

Recordemos que, en la solución de vuelta atrás del Ejercicio 14.21, un franqueo admisible se representaba como una tupla $(x_1,\dots,x_5)$ donde xi es el tipo de sello que se ha colocado en la casilla *i.*

Un franqueo (parcial) será factible solo si no se utilizan más de 3 sellos de cada tipo. También hay que tener en cuenta que queden suficientes sellos para rellenar todas las casillas y que tengan valor suficiente para cubrir la tarifa restante. Por otra parte, como lo importante son los valores escogidos y no las casillas actuales que ocupan, se impone un orden en la colocación de los sellos.

En cuanto a las estimaciones a utilizar para el esquema optimista-pesimista, tenemos:

optimista: Una cota inferior muy sencilla es rellenar las casillas restantes con los sellos

de menor valor, aunque esta estimación nunca debe ser menor que la tarifa a cubrir.

pesimista: Dualmente, se obtiene una cota superior rellenando las casillas restantes con los sellos de mayor valor.

Nos interesa, por tanto, tener ordenados los tipos de sellos según su valor de menor a mayor.

Se puede combinar el cálculo de la estimación pesimista con la comprobación de factibilidad, ya que si esta estimación superior no cubre la tarifa solicitada, es que no se puede cubrir con los sellos que quedan. De este modo, aunque por debajo de un nodo puede, en general, no haber soluciones, solo se utiliza la estimación pesimista cuando se sabe que sí las hay.

En cada nodo, además de la información usual (solución parcial, etapa y prioridad), guardamos el último tipo de sellos considerado, junto con el número de sellos utilizados de este tipo, y la tarifa acumulada.

```
tipos
  nodo = reg
        sol[1..5] de 1..n
        k: 0..5
        coste: real
        costeOpt: real   {prioridad}
        último: 1..n
        usados: 0..3
      freg
ftipos
```

Además, si se llega a alcanzarla tarifa *T*

de forma exacta, podremos terminar inmediatamente la búsqueda. Como *costeMejor* puede no corresponder a la mejor solución encontrada, sino a una estimación pesimista muy ajustada, no podemos simplemente comparar *costeMejor = T.* Añadimos una variable *encontrada* que nos indique tal circunstancia.

El algoritmo es el siguiente:

```
//{V[1]≤...≤V[n]}
fun sellosRp(V[1..n] de real⁺, T: real⁺) dev
⟨solMejor[1 ..5] de 1..n, costeMejor: real⟩
var X, Y: nodo, C: colapr[nodo]

encontrada := falso;
//{generamos la raíz}
Y.k := 0;
Y.coste := 0;
Y.último := 1;
Y.usados := 0;
⟨Y.costeOpt, costeMejor⟩ :=
cálculoEstimaciones(T, V, Y.k, Y.usados. Y.coste,
Y.último);
C := cpVacía();
añadir(C, Y);
mientras ¬encontrada ∧ ¬esCpVacía?(C) ∧
mínimo(C).costeOpt ≤ costeMejor hacer
    Y := mínimo(C);
    eliminarMín(C);
    //{generamos los hijos de Y}
    X.k := Y.k + 1;
    X.sol := Y.sol;
    {probamos un sello de tipo Y.último}
    si Y.usados < 3 entonces
        X.último := Y.último;
        X.sol[X.k] := X.último;
        X.usados := Y.usados + 1;
        X.coste := Y.coste + V[X.último];
        ⟨X.costeOpt, pes⟩ := cálculoEstimaciones(T, V,
        X.k, X.usados, X.coste, X.último);
        si X.costeOpt ≤ costeMejor entonces
```

```
si X.k = 5 entonces
si X.coste ≥ T entonces  //{es solución}
solMejor := X.sol;
costeMejor := X.coste;
encontrada := (X.coste = T) //{terminar}
fsi
si no
añadir(C, X);
costeMejor := min(costeMejor, pes);
fsi
fsi
fsi

//{probamos con el resto de tipos de sellos}
X.usados := 1;
s := Y.último + 1;
mientras ⌐encontrada ∧ s ≤ n hacer
X.último := s;
X.sol[X.k] := s;
X.coste := Y.coste + V[s];
⟨X.costeOpt, pes⟩ := cálculoEstimaciones(T, V,
X.k, X.usados, X.coste, X.último);
si X.costeOpt ≤ costeMejor entonces
si X.k = 5 entonces
si X.coste ≥ T entonces  //{es solución}
solMejor := X.sol;
costeMejor := X.coste;
encontrada := (X.coste = T)  //{terminar}
fsi
si no
añadir(C, X);
costeMejor := mín(costeMejor, pes);
fsi
fsi
s := s+1;
fmientras
fmientras
ffun
```

Para calcular las estimaciones, primero obtenemos la pesimista (cota superior), que nos sirve para ver si el nodo es factible; solo en caso afirmativo se calcula la optimista.

{V[1] ≤ … ≤ V[n]}

```
fun cálculoEstimaciones(V[1..n] de real+, T:
real+, k: 0..5, usados: 0..3, coste: real, último:
1..n) dev ⟨opt, pes: real∞ ⟩

//{rellenar con los sellos de más valor}
usadosAux := 0;
costeAux := coste;
j := k+1;
i := n;
mientras j ≤ 5 ∧ i ≥ último hacer
si usadosAux < 3 entonces
usadosAux := usadosAux + 1;
costeAux := costeAux + V[i];
j := j + 1;
si no
i := i – 1;
si i > último entonces usadosAux := 0
si no usadosAux := usados;
fsi
fsi
fmientras
si costeAux < T  ∨ j ≤ 5 entonces
//{no es factible: no quedan suficientes sellos o no
pueden cubrir la tarifa}
⟨opt, pes⟩ := ⟨+∞, +∞⟩;
si no
pes := costeAux;
//{rellenar con los sellos de menor valor}
usadosAux := usados;
costeAux := coste;
j := k + 1;
i := último;
mientras j ≤ 5 ∧ i ≤ n hacer
si usadosAux < 3 entonces
usadosAux := usadosAux + 1;
costeAux := costeAux + V[i];
j := j + 1;
si no i := i+1; usadosAux := 0;
fsi
fmientras
opt := máx(T, costeAux);
fsi
ffun
```

15.8. Recorrido barato

Un vendedor tiene que recorrer *n* ciudades volviendo tras ello al punto de partida. Se ha informado sobre las posibles conexiones directas por ferrocarril entre las ciudades y sobre las tarifas correspondientes.

El vendedor desea conocer un circuito en tren que recorra cada ciudad exactamente una vez y regrese a la ciudad de partida, y cuya tarifa total sea mínima.

-------------------Solución----------------

Este problema se resolvió en el Apartado (b) del *Ejercicio 14.14* mediante la técnica de *vuelta atrás*, representando la información medíante un grafo dirigido y valorado donde los vértices son las ciudades (numeradas de 1 a *n) y* las aristas son las conexiones directas de ferrocarril, valoradas con la tarifa correspondiente.

Recordemos que las soluciones son tuplas $(x_1,\dots,x_n)$, donde x_i es la ciudad por la que pasamos en i-ésimo lugar. Para evitar soluciones repetidas, fijamos el comienzo de los circuitos en la ciudad 1.

Se vio cómo calcular una estimación inferior del coste real de una solución utilizando el mínimo global de los valores de todas las aristas del grafo para aproximar el coste de las conexiones que faltan por realizar.

Este problema puede no tener solución, lo que representamos con un coste $+\infty$. Por tanto, no podemos utilizar una cota superior del coste de una

solución parcial, porque no se garantiza que exista alguna solución en el correspondiente subárbol. Por esta razón, en este ejercicio ***no*** podemos utilizar una cota pesimista, y *coste-mejor* se actualizará solo al encontrar una solución (esquema de la Sección 15.2).

En cada nodo, además de la información usual (solución parcial, etapa y prioridad) llevamos cuenta del coste acumulado y de las ciudades ya visitadas:

```
tipos
  nodo = reg
        sol[1 ..n] de 1..n
        k: 1..n
        coste: real;
        costeEstimado: real    {prioridad}
        usado[1..n] de bool
      freg
ftipos
```

El algoritmo es el siguiente:

```
fun vendedorRp(G: grafoVal[n]) dev
⟨solMejor[1..n] de 1..n, costeMejor: real∞⟩
var X, Y: nodo, C: colapr[nodo]

mínG := cálculoMínimo(G);
{generamos la raíz}
Y.k : = 1;
Y.sol[1] := 1;
Y.usado[1] := cierto;
Y.usado[2..n] := [falso];
Y.coste := 0;
Y.costeEstimado := n * mínG;
C := cpVacía();
añadir(C, Y);
costeMejor := +∞; {para indicar que no hay
solución}
mientras ¬esCpVacía?(C) ∧
minimo(C).costeEstimado < costeMejor hacer
```

```
Y := mínimo(C);
eliminarMín(C);
{generamos los hijos de Y}
X.k := Y.k + 1;
X.sol := Y.sol;
X.usado := Y.usado;
anterior := X.sol[X.k - 1]
para vértice = 2 hasta n hacer
  si ⌐X.usado[vértice] ∧ gvEstáArista?(anterior,
  vértice, G) entonces
    X.sol[X.k] := vértice;
    X.usado[vértice] := cierto;
    X.coste := Y.coste + gvValor(anterior, vértice,
    G);
    si X.k = n entonces
      si gvEstáArista?(sol[n], 1, G) ∧c X.coste +
      gvValor(X.sol[n], 1, G) < costeMejor
      entonces
        solMejor := X.sol;
        costeMejor := X.coste + gvValor(X.sol[n],
        1, G);
      fsi
    si no
      X.costeEstimado := X.coste + (n - X.k + 1)
      * mínG;
        si X.costeEstimado < costeMejor
        entonces añadir(C, X);
        fsi
    fsi
    X.usado[vértice] := falso;
  fsi
fpara
fmientras
ffun
```

La función cálculoMínimo que calcula el mínimo global de los valores de todas las aristas ya está definida para la solución con la técnica de *vuelta atrás* en el *Ejercicio 14.14*.

15.9. Empaquetamiento óptimo.

Se tiene una colección de *n* objetos “moldeables”, cada uno con un volumen vi, para *i* entre 1 y *n,* que hay que empaquetar utilizando envases de capacidad *E.* Diseñar un algoritmo que calcule el empaquetamiento óptimo, es decir, que minimice la cantidad de envases utilizados, teniendo en cuenta que los objetos **no se pueden fraccionar**.

----------------Solución-----------------

Recordemos que en la solución de *vuelta atrás* (Ejercicio 14.22) se supone que todo objeto cabe en un envase vacío y, por tanto, se necesitan un máximo de *n* envases. Las soluciones se representan en tuplas de la forma $(x_1,\dots,x_n)$ donde xi es el envase donde hemos colocado el objeto i. Como los envases son indistinguibles, el primer objeto siempre se coloca en el primer envase ($x_1=1$) y para cada objeto de los restantes se puede usar uno de los envases ya ocupados, si cabe en alguno, o coger uno vacío.

En cuanto a las estimaciones a utilizar para el esquema optimista-pesimista, tenemos:

- **optimista:** Una cota inferior adecuada es el número de envases ya utilizados en la solución parcial.
- **pesimista:** Una cota superior muy sencilla es considerar un envase extra por cada objeto que nos

queda por empaquetar, pero resulta demasiado grosera. Podemos en cambio ir considerando cada objeto restante, en el orden que se haya dado, e intentar meterlo en el primer envase utilizado y, en el caso de que no quepa, intentarlo con el segundo envase, y así hasta agotar todas las posibilidades, en cuyo caso se añadirá un nuevo envase a la solución parcial.

En cada nodo, además de la información usual (solución parcial, etapa y prioridad), guardamos el número de envases utilizados y las capacidades disponibles de cada envase utilizado.

tipos
  nodo = **reg**
    sol[1..n] de 1..n
    k: 1..*n*
    envases: 1..n {prioridad}
    capacidad[1..n] **de** *real*
  freg
ftipos

Además, como en la solución mediante la técnica de *vuelta atrás* (Ejercicio 14.22), la búsqueda podrá acabarse si encontramos una solución con el valor

$$\text{óptimo} = \left\lceil \frac{\sum_{i=1}^{n} v_i}{E} \right\rceil.$$

El algoritmo es el siguiente:

{∀i: 1≤i≤*n* : V[i]≤*E*}

fun **empaquetarRp**(E: *real*⁺, V[1..n] **de** *real*⁺)
dev ⟨*solMejor*[1 ..n] de 1..n, *envasesMejor*: 1..*n*⟩
var *X*, *Y*: *nodo*, *C*: *colapr*[*nodo*]

```
total := 0
para i = 1 hasta n hacer
    total := total + V[i];
fpara
óptimo := ⌈ total/E ⌉;
encontrada := falso;
{generamos la raíz: el primer objeto en el primer envase}
Y.k := 1;
Y.sol[1] := 1;
Y.envases := 1;
Y.capacidad[1] := E - V[1];
Y.capacidad[2..n] := [E];
C := cpVacía();
añadir(C, Y);
envasesMejor := cálculoPesimista(E, V, Y);
mientras ¬encontrada ∧ ¬esCpVacía?(C) ∧
mínimo(C).envases ≤ envasesMejor hacer
    Y := mínimo(C);
    eliminarMín(C);
    {generamos los hijos de Y}
    X.k := Y.k + 1;
    X.sol := Y.sol;
    X.envases := Y.envases;
    X.capacidad := Y.capacidad;
    {probamos con cada envase ya utilizado}
    i := 1;
    mientras i ≤ Y.envases ∧ ¬encontrada hacer
        si X.capacidad[i] ≥ V[X.k] entonces
            X.sol[X.k] := i;
            X.capacidad[i] := X.capacidad[i] - V[X.k];
            si X.k = n entonces
                solMejor := X.sol;
                envasesMejor := X.envases;
                encontrada := (envasesMejor = óptimo);
                {terminar}
            si no
                añadir(C, X);
                pes := cálculoPesimista(E, V, X);
                envasesMejor := min(envasesMejor, pes);
            fsi
            X.capacidad[i] := Y.capacidad[i];
        fsi
    fmientras
```

```
si ¬encontrada entonces
  {probamos con un envase nuevo}
  nuevo := Y.envases + 1;
  X.sol[X.k] := nuevo;
  X.envases := nuevo;
  X.capacidad[nuevo] := E – V[X.k];
  si X.envases ≤ envasesMejor entonces
    si X.k = n entonces
      solMejor := X.sol;
      envasesMejor := nuevo;
      encontrada := (envasesMejor = óptimo);
      {terminar}
    si no
      añadir(C, X);
      pes := cálculoPesimista(E, V, X);
      envasesMejor := min(envasesMejor, pes);
    fsi
  fsi
fsi
fmientras
ffun
```

Para calcular las estimaciones usamos la siguiente función:

```
fun cálculoPesimista(E: real+, V[1..n] de real+, X:
nodo) dev pes: 1..n
var capacidadAux[1 ..n] de real

pes := X.envases;
capacidadAux := X.capacidad;
para i = X.k + 1 hasta n hacer
  j := 1;
  mientras V[i] > capacidadAux[j] hacer
    j := j + 1;
  fmientras
  capacidadAux[j] := capacidadAux[j] - V[i];
  pes := max(pes, j);
fpara
ffun
```

BIBLIOGRAFÍA

A continuación ofrecemos una selección de textos que consideramos especialmente recomendables para los estudiantes de asignaturas con contenidos de estructuras de datos y métodos algorítmicos. No incluimos aquí muchas otras obras que aunque son de gran calidad e interés se alejan o bien del enfoque o bien del nivel que hemos seguido en este libro. (En particular no consideramos la gran cantidad de libros cuyo objetivo es programar algoritmos o estructuras de datos en un lenguaje concreto).

[BB90] **Brassard y Bratley.** *Algorítmica. Concepción y análisis.* Masson. **1990.**

Este texto es un predecesor del publicado años más tarde por los mismos autores [BB97], Aunque la nueva versión mejora a esta en muchos aspectos, tanto metodológicos como de contenidos, sigue siendo una referencia obligada en el diseño de algoritmos porque aparecen aquí algunos algoritmos y soluciones interesantes que han sido excluidos del texto nuevo.

[BB97] **Brassard** y **Bratley.** *Fundamentos de algoritmia.* Prentice Hall. 1997.

Es uno de los libros de texto sobre algoritmos más recomendable porque ofrece una visión suficientemente amplia (sin ser exhaustiva) de los métodos algorítmicos y la implementación de las estructuras de datos. En general, las explicaciones son claras y fácilmente comprensibles por los estudiantes. Los algoritmos se expresan en un pseudocódigo tipo Pascal del mismo estilo que el utilizado en el presente libro.

[CLRS011 **Cornien, Leiserson, Rivest y Stein.** *Introduction to algorithms. Second edition.* The MIT Press. 2001.

La palabra "*introducción*" en su título no debe llamarnos a engaño, pues es un volumen de 1180 páginas repletas de teoremas que demuestran la corrección de los algoritmos presentados *y* fórmulas que justifican la complejidad del coste de los mismos. Tampoco se trata de un libro dedicado solamente al diseño de algoritmos, sino que incluye múltiples temas relacionados con las estructuras de datos. Como libro de texto para el alumno resulta un tanto árido, pero es un excelente libro de consulta porque abarca muchos temas que no suelen aparecer en otros libros de algoritmos, todos

ellos tratados con gran profundidad y con mucho rigor.

[Fra94] **Xavier Franch Gutiérrez,** *Estructuras de datos. Especificación, diseño e implementación.* Edicions UPC, 1994.
Como su título indica, en este texto se tratan las estructuras de datos desde sus tres aspectos fundamentales. Las especificaciones son algebraicas, del estilo de las que aquí ofrecemos, y las implementaciones se realizan en pseudocódigo. Se pone especial cuidado en resaltar la relación entre una especificación y sus posibles implementaciones. Solo trata las estructuras de datos más típicas, pero con gran detalle y acertadas explicaciones.

[Har92] **Harel.** *Algorithmics. The spirit of computing. Second edition.* Addison VVesley. 1992.
Es un libro de obligada lectura para cualquier persona interesada en el mundo de la computación. No se trata de un libro de texto, ni de una obra de consulta: podríamos decir que es un libro de "divulgación algorítmica", porque introduce todos los temas relacionados con la algoritmia (lenguajes de programación, métodos, corrección, eficiencia, limitaciones, etc.) de una manera muy intuitiva y amena, de forma que el libro debe leerse de principio a fin como si de una novela se tratara.

[HS94] Horowitz y Sahni. *Fundamentáls of data structures in Pascal. Fourth edition.* Computer Science Press, 1994.
Esta es la última edición (en su versión en Pascal, pues existen versiones en otros lenguajes) de un texto ya clásico sobre estructuras de datos. Contiene numerosas y variadas implementaciones de toda clase de estructuras, incluyendo muchas variaciones de estructuras arbóreas de búsqueda y de estructuras basadas en montículos. Es una lástima que algunas explicaciones no sean todo lo claras que uno desearía.

[HSR98] Ellis Horowitz, Sartaj Sahni y Sanguthevar Rajasekaran. *Computer algorithms.* Computer Science Press. 1998.
Este texto es sucesor del también clásico

Fundamentáis of Computer algorithms escrito en el año 1978 por los dos primeros autores sobre métodos algorítmicos, siendo el complemento perfecto de [HS94]. Contiene todos los algoritmos más ''típicos" y muchísimos más, todos ellos escritos en pseudocódigo tipo C++. Cada algoritmo viene acompañado de una exhaustiva justificación de su corrección. Este texto trata también varios temas más allá del nivel del presente libro.

[Lev03] **Anany Levitin.** *Introduction to the design and analysis of algorithms*. Addison Wesley, 2003.'

Este reciente libro de texto sobre métodos algorítmicos merece ser incluido en esta selección por su novedosa (aunque discutible) clasificación de las técnicas de diseño de algoritmos. La presentación de los contenidos es altamente pedagógica, pero poco profunda En muchas ocasiones los algoritmos solamente se describen con palabras y en otras se expresan en un pseudocódigo muy' abstracto, dejando demasiados detalles sin resolver. El texto incluye en total unos 600 ejercicios para la mayoría de los cuales el autor da algunas indicaciones en un apéndice.

[Man89] **Udi Manber.** *Introduction to algorithms. A Creative approach*. Addison-Wesley, 1989.

Este texto resulta bastante original en su planteamiento, que consiste en "derivar" los algoritmos a partir de un razonamiento inductivo, de forma que la corrección del algoritmo queda justificada antes de expresarlo en un pseudocódigo tipo Pascal. Se centra en el diseño de los algoritmos, y el único capítulo dedicado a las estructuras de datos resulta muy escaso; por contra, el capítulo dedicado a los grafos es muy completo. Aunque no es válido como libro de texto para los alumnos, es una valiosa fuente de ejemplos y ejercicios adicionales.

[NN98] Richard Neapolitan y Kumarss Naimipour. *Foundations of algorithms using C++ pseudocode. Second edition.* Jones and Bartlett Publishers. 1998.

Esta obra también resulta muy recomendable

para los alumnos como libro de texto sobre métodos algorítmicos porque la exposición de los lemas y las explicaciones resultan muy comprensibles. Sin embargo, los contenidos quedan un tanto escasos y resulta imprescindible completarlos con otros textos.

[Par95] **Parberry.** *Problems on algorithms*. Prentice Hall. 1995.

Este pequeño texto es una colección de ejercicios sobre el diseño, la corrección y el análisis de los algoritmos (aunque también contiene un capítulo, muy reducido, sobre estructuras de datos). Para algunos (demasiado pocos) de estos ejercicios el autor ofrece soluciones y para algunos otros da ciertas indicaciones. Esta colección ha sido fuente de inspiración de varios problemas para el presente libro.

[Peñ98] **Peña.** *Diseño de programas. Formalismo y abstracción. Segunda edición* Prentice Hall, 1998.

Este libro es muy recomendable como libro de texto para los alumnos que estudian tipos abstractos de datos y diseño de algoritmos desde el punto de vista de la especificación y la corrección de los mismos, sin abordar el tema de los métodos algorítmicos. Es uno de los pocos textos que tratan la especificación de los tipos abstractos de datos formal y rigurosamente.

También aporta implementaciones de los tipos abstractos de datos, pero carece de aplicaciones de estas estructuras. Para cada tema se proponen unos pocos ejercicios, incluyéndose al final del texto soluciones para la mitad de ellos.

[Ski98] **Skiena.** *The algoríthm design manual*. TELOS Pr. 1998.

Este texto está dividido en dos partes. La primera está dedicada a los métodos algorítmicos y aunque la presentación de cada técnica no sea muy completa, resultan muy interesantes las “batallitas" del autor, que son aplicaciones prácticas experimentadas por él mismo. Pero lo más original del libro es la segunda parte, que contiene un catálogo muy completo de problemas algorítmicos. Para cada problema, además del enunciado y posibles aplicaciones prácticas, se proporciona

información sobre los algoritmos más conocidos para su resolución. Si bien dichos algoritmos no se detallan en el texto, se dan abundantes referencias para encontrarlos. con indicación de los lenguajes en que han sido implementados y consideraciones sobre los costes conseguidos. El libro viene acompañado de un CD-ROM que contiene el código fuente de todos los algoritmos citados, una versión hipertexto de todo el libro y otro material docente adicional.

[WeiOO] **Mark Allen Weiss.** *Estructuras de datos en Java*. Addison Wesley. 2000.

El interés de este libro reside en su visión más práctica de las estructuras de datos, ofreciendo implementaciones detalladas y numerosas aplicaciones de estructuras tanto básicas como avanzadas utilizando el lenguaje Java. Al utilizar técnicas orientadas a objetos introduce el concepto de tipo abstracto de datos haciendo hincapié en los principios de ocultación y encapsulación. pero sin abordar el tema de la especificación.

Estructuras de datos y métodos algorítmicos

Ejercicios resueltos

Este manual surge con la intención de ofrecer una colección de ejercicios resueltos sobre estructuras de datos y métodos algorítmicos.

El libro se divide en dos partes:

- La primera se dedica a las estructuras de datos. Empieza con sendos capítulos genéricos sobre especificación algebraica de tipos abstractos de datos y su implementación, para continuar con capítulos dedicados a las estructuras de datos más habituales: pilas, colas, listas, árboles binarios y generales, árboles de búsqueda y tablas, colas de prioridad y montículos, así como grafos. Esta parte termina con un capítulo de aplicaciones donde se usan las estructuras anteriores de diferentes formas.

- La segunda parte, la de algoritmos, se organiza alrededor de la clasificación habitual de métodos algorítmicos: divide y vencerás, método voraz, programación dinámica, vuelta atrás y ramificación y poda.

El lenguaje utilizado a lo largo del libro a la hora de escribir las especificaciones y los programas es un lenguaje abstracto para ambos aspectos, de forma que se evitan algunos detalles demasiado concretos que suponen la utilización de un lenguaje particular y se puede poner el énfasis de las soluciones en un nivel de abstracción superior.

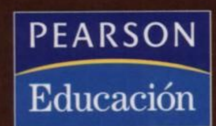


www.prenticepractica.com

PRENTICE PRACTICA es una colección de libros, cuyo texto es eminentemente práctico. La finalidad de esta colección es permitir al alumno comprender y afianzar la asimilación de la teoría a través de diversos ejercicios y ejemplos.

PRENTICE PRACTICA es una colección amena, de carácter muy didáctico y que, de una forma sencilla, consigue que el alumno obtenga un perfecto manejo práctico de la asignatura.

PRENTICE PRACTICA está dirigida al alumno para conseguir su autoaprendizaje en la materia. La colección es una de las más actualizadas del mercado.



www.pearsoneducacion.com

ISBN 84-205-3849-3

